

Heuristics to predict and eagerly translate code in DBTs

Surya Tej Nimmakayala

Submitted to the graduate degree program in Electrical
Engineering and Computer Science and the Graduate Faculty
of the University of Kansas School of Engineering in partial
fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Committee:

Dr. Prasad Kulkarni: Chairperson

Dr. Perry Alexander

Dr. Bo Luo

Dr. Fengjun Li

Dr. Shawn Keshmiri

Date Defended

The Thesis Committee for Surya Tej Nimmakayala certifies
That this is the approved version of the following thesis:

Heuristics to predict and eagerly translate code in DBTs

Committee:

Chairperson

Date Approved

Acknowledgements

I am grateful to my parents support and patience throughout this long endeavor.

I would like to thank all the Professors under whom I have taken my courses, which helped in gaining exposure to the areas necessary to do my research. It also gave me a better understanding of computer science as a whole and how the different fields have the potential in taking the existing technology to the next level.

Abstract

Dynamic Binary Translators(DBTs) have a variety of uses, like instrumentation, profiling, security, portability, etc. In order for the desired application to run with these enhanced additional features(not originally part of its design), it is to be run under the control of Dynamic Binary Translator. The application can be thought of as the guest application, to be run with in a controlled environment of the translator, which would be the host application. That way, the intended application execution flow can be enforced by the translator, thereby inducing the desired behavior in the application on the host platform(combination of Operating System and Hardware).

However, there will be a run-time/execution-time overhead in the translator, when performing the additional tasks to run the guest application in a controlled fashion. This run-time overhead has been limiting the usage of DBT's on a large scale, where response times can be critical. There is often a trade-off between the benefits of using a DBT against the overall application response time. So, there is a need to research/explore ways to faster application execution through DBT's(given their large code-base).

With the evolution of the multi-core and GPU hardware architectures, multiple concurrent threads can get more work done through parallelization. A proper design of parallel applications or parallelizing parts of existing serial code, can lead to improved application run-time's through hardware architecture support.

We explore the possibility of improving the performance of a DBT named DynamoRIO. The basic idea is to improve its performance by speeding-up the process of guest code translation, through multiple threads translating multiple pieces of code concurrently. In an ideal case, all the required code blocks for application execution are readily available ahead of time without any stalls. For efficient eager translation, there is also a need for heuristics to better predict

the next code block to be executed. That could potentially bring down the less productive code translations at run-time. The goal is to get application speed-up through eager translation and block prediction heuristics, with execution time close to native run.

Contents

Abstract	iii
Table of Contents	v
List of Figures	vii
List of Tables	x
1 Introduction	1
2 Related Work	3
3 Background	6
3.1 Role of Dynamic Binary Translator(DBT)	6
3.2 High-Level Design and Optimizations of DynamoRIO	7
3.2.1 Basic Block Cache	8
3.2.2 Trace Cache	9
3.2.3 Linking Direct Branches	10
3.2.4 Indirect Branch Inlining	10
4 Exploring Run-Time Bottlenecks	13
4.1 Benchmarks and Environment set-up for experiments	13
4.2 Supporting Experimental Results and Analysis	14
4.2.1 Overhead Measure: Run-times and Compilation-times . . .	14
4.2.2 Code translations and Control Exiting Code-Cache	17
5 Eager Translation: Experimental Set-Up and Results	21
5.1 Multiple compiler threads	22

5.2	Simulation of multiple compiler threads	24
6	Need for a BB Prediction Heuristic	36
7	Generation and Application of Heuristics	40
7.1	Heuristics Through Data Mining	40
7.2	Input Data for LEM1	43
7.3	Applying Heuristics in DynamoRIO	46
7.3.1	In-Domain vs Out-Domain Heuristics	48
8	Eager Translation with Heuristics: Experimental Set-Up and Results	50
8.1	Experimental Set-Up	50
8.1.1	Extract Basic Block Input data for LEM1	51
8.1.2	Applying Heuristics During Eager Translation	53
8.1.2.1	In-Domain vs Out-Domain	54
8.2	Analysis of Experimental data	54
8.2.1	In-Domain	57
8.2.2	Out-Domain	77
8.3	Summary	86
9	Future Work	91
10	Conclusion	93
	References	97

List of Figures

3.1	DynamoRIO execution flow from [11]	8
4.1	Ratio of <i>DynamoRIO</i> run times to <i>Native</i> run times for different benchmark <i>test inputs</i>	15
4.2	Ratio of <i>DynamoRIO</i> run times to <i>Native</i> run times for different benchmark <i>reference inputs</i>	16
4.3	Run-Time overhead distribution: total/app. exec. time/compilation time, compared to native run-time.	17
4.4	Distribution of fcache exit causes for different benchmark test inputs	19
4.5	Distribution of instruction counts for different execution phases of 456.hmmer benchmark input bombesin	19
4.6	Distribution of instruction counts for different execution phases of 400.perlbench benchmark input regmesg.pl	20
5.1	Basic block count captured with static list of addresses(<i>or without eager translation</i>), for different thread configurations.	24
5.2	Application run-time with static list of addresses(<i>or without eager translation</i>), for different thread configurations.	25
5.3	App Run-Time <i>without eager translation</i> for <i>simulated</i> compiler thread counts: 1,2,4,8.	26
5.4	App Run-Time <i>without eager translation</i> for <i>simulated</i> compiler thread counts: 16,32,64,128.	27
5.5	App Run-Time with <i>Eager Translation</i> for <i>simulated</i> compiler thread counts: 1,2,4,8.	28
5.6	App Run-Time with <i>Eager Translation</i> for <i>simulated</i> compiler thread counts: 16,32,64,128.	29

5.7	App Run-Time with and without <i>Eager Translation</i> for <i>simulated</i> compiler thread counts: <i>virtually infinite or 10000</i>	30
5.8	Requested BB translations without <i>Eager Translation</i> for <i>simulated</i> thread configurations: <i>1, 2, 4, 8, 16</i>	31
5.9	Requested BB translations without <i>Eager Translation</i> for <i>simulated</i> thread configurations: <i>32, 64, 128</i>	32
5.10	Requested BB translations with <i>Eager Translation</i> for <i>simulated</i> thread configurations: <i>1, 2, 4, 8</i>	32
5.11	Requested BB translations with <i>Eager Translation</i> for <i>simulated</i> thread configurations: <i>16, 32, 64, 128</i>	33
5.12	Requested BB translations with and without <i>Eager Translation</i> for different number of <i>simulated</i> compiler threads.	34
6.1	Fraction of useful eager translations for different benchmarks and different number of <i>simulated</i> compiler threads.	38
8.1	Comparison of run-time stats for the test input runs, applying <i>certain</i> rule-sets with <i>eager translation</i>	58
8.2	Comparison of run-time stats for the test input runs, applying <i>possible</i> rule-sets with <i>eager translation</i>	60
8.3	Comparison of run-time stats for the test input runs, applying <i>common</i> rule-sets with <i>eager translation</i>	61
8.4	<i>Urgent Request Drop Count, Explicit Eager Translation Count</i> stats when applying <i>certain</i> rule-sets with <i>eager translation</i>	64
8.5	<i>Urgent Request Drop Count, Explicit Eager Translation Count</i> stats when applying <i>possible</i> rule-sets with <i>eager translation</i>	66
8.6	<i>Urgent Request Drop Count, Explicit Eager Translation Count</i> stats when applying <i>common</i> rule-sets with <i>eager translation</i>	68
8.7	Comparison of <i>% Useful Eager Translation</i> and <i>% Urgent Requests</i> for the test vs ref inputs applying <i>certain</i> rule-sets with <i>eager translation</i>	80
8.8	Comparison of <i>% Useful Eager Translation</i> and <i>% Urgent Requests</i> for the test vs ref inputs applying <i>possible</i> rule-sets with <i>eager translation</i>	82

8.9	Comparison of % <i>Useful Eager Translation</i> and % <i>Urgent Requests</i> for the test vs ref inputs applying <i>common</i> rule-sets with <i>eager</i> <i>translation</i>	83
-----	---	----

List of Tables

4.1	Different stats related to the benchmark test inputs	18
6.1	Computation of fraction useful eager translation for different benchmarks run with 10000 <i>simulated</i> threads.	38
7.1	Sample data-set to decide on a <i>Trip</i> based on <i>Wind, Humidity, Temperature</i>	42
7.2	Sample records with <i>basic block</i> data attributes and <i>branch prediction</i> decision attribute: <i>isDCBTaken</i>	45
8.1	<i>Old Attr. vs New Attr.</i> consolidated run-time stats for all <i>SPEC 2006</i> benchmark in-domain runs with their respective rule-sets . .	62
8.2	Multi-input benchmark run configuration: Rule-set of one input applied to another in same benchmark	71
8.3	% <i>Useful Etrans. stat ratios</i> (with respective rule-set stats) for <i>SPEC 2006</i> benchmark runs: <i>SPEC_INT</i> logical in-domain with certain rule-set	73
8.4	% <i>Urgent Request stat ratios</i> (with respective rule-set stats) for <i>SPEC 2006</i> benchmark runs: <i>SPEC_INT</i> logical in-domain with certain rule-set	74
8.5	<i>Run-Time Overhead stat ratios</i> (with respective rule-set stats) for <i>SPEC 2006</i> benchmark runs: <i>SPEC_INT</i> logical in-domain with certain rule-set	75
8.6	<i>Consolidated stat ratios</i> (with respective rule-set stats) for <i>SPEC 2006</i> benchmark runs: <i>SPEC_INT</i> logical in-domain with certain rule-set	76

8.7	% <i>Useful Etrans. stat ratios</i> (with respective rule-set stats) for <i>SPEC 2006</i> benchmark runs: <i>SPEC_FLOAT</i> logical in-domain with certain rule-set	77
8.8	% <i>Urgent Request stat ratios</i> (with respective rule-set stats) for <i>SPEC 2006</i> benchmark runs: <i>SPEC_FLOAT</i> logical in-domain with certain rule-set	78
8.9	<i>Run-Time Overhead stat ratios</i> (with respective rule-set stats) for <i>SPEC 2006</i> benchmark runs: <i>SPEC_FLOAT</i> logical in-domain with certain rule-set	78
8.10	<i>Consolidated stat ratios</i> (with respective rule-set stats) for <i>SPEC</i> <i>2006</i> benchmark runs: <i>SPEC_FLOAT</i> logical in-domain with cer- tain rule-set	79
8.11	<i>Consolidated stat ratios</i> (with respective rule-set stats) for <i>SPEC</i> <i>2006</i> benchmark runs: <i>SPEC_INT</i> logical out-domain with certain rule-set	84
8.12	% <i>Useful Etrans. stat ratios</i> (with respective rule-set stats) for <i>SPEC 2006</i> benchmark runs: <i>SPEC_INT</i> logical out-domain with certain rule-set	85
8.13	% <i>Urgent Request stat ratios</i> (with respective rule-set stats) for <i>SPEC 2006</i> benchmark runs: <i>SPEC_INT</i> logical out-domain with certain rule-set	85
8.14	<i>Run-Time Overhead stat ratios</i> (with respective rule-set stats) for <i>SPEC 2006</i> benchmark runs: <i>SPEC_INT</i> logical out-domain with certain rule-set	86
8.15	<i>Consolidated stat ratios</i> (with respective rule-set stats) for <i>SPEC</i> <i>2006</i> benchmark runs: <i>SPEC_FLOAT</i> logical out-domain with cer- tain rule-set	87
8.16	% <i>Useful Etrans. stat ratios</i> (with respective rule-set stats) for <i>SPEC 2006</i> benchmark runs: <i>SPEC_FLOAT</i> logical out-domain with certain rule-set	88
8.17	% <i>Urgent Request stat ratios</i> (with respective rule-set stats) for <i>SPEC 2006</i> benchmark runs: <i>SPEC_FLOAT</i> logical out-domain with certain rule-set	89

8.18	<i>Run-Time Overhead stat ratios</i> (with respective rule-set stats) for <i>SPEC 2006</i> benchmark runs: <i>SPEC_FLOAT</i> logical out-domain with certain rule-set	90
------	---	----

Chapter 1

Introduction

Dynamic Binary Translation Systems (DBTs) provide a controlled execution environment for an application to run. DBT can be considered as a *host* and the executed application as *guest*. To retain control over the application, the original source code is first translated and cached for future use, which is then executed instead of the original code. So, the actual source binary is never executed. Only the translated code. Since DBT does the translation, it can instrument the code blocks to ensure control and valid flow. This privileged control also opens up useful applications like profiling [4, 19], program optimization [1], binary portability [2, 21, 22] and secure execution [16, 18].

Even though the primary task of DBTs is the translation of guest code to host format, it still needs to perform auxilliary tasks that cause run-time overhead, like resolving indirect branches and so forth. Instrumentation done to induce characteristics that are not originally part of the guest program can increase the overhead, at which point it would be a trade-off and can limit the use of DBTs. Many optimization techniques have been introduced to amortize the run-time overhead in DBTs like, code cache for translated code [20], direct branch chaining [5], in-

direct branch prediction [15] and traces [1]. In spite of these techniques there is still a significant overhead hindering the wide-spread use of DBTs, especially with short-running programs.

We have done some preliminary work in exploring the causes of bottlenecks with a DBT named DynamoRIO (discussed in chapter 3) and found that control exiting code cache due to block translation and indirect branches are prominent factors. It will be discussed in chapter 4. Then we present an approach called eager translation in chapter 5 to address the overhead being caused by block translation. Idea is to translate code ahead of time through multiple threads such that the application can have a more fluid execution with control staying more within the code cache through block linking. More details along with related findings and challenges to eager translation will also be discussed in chapter 5. After that the role of *heuristics* in *eager translation* and how they have been generated and applied will be discussed in final chapters 6, 7, 8.

Related work on finding plausible reasons for overhead will be discussed next in chapter 2.

Chapter 2

Related Work

Earlier exploration for bottlenecks in Dynamic Binary Translation Systems(DBTs) was done by researchers. As mentioned in prior chapter, DBTs are complex systems with big code bases designed to perform additional tasks apart from the source to target code translation. The extra work done during application run causes overhead depending not only on the interactions between components of the DBT system but also the computer system/platform that enables the use of DBT. Impact of application runs under DBTs on computer micro-architecture has been researched by Arkaitz Ruiz and Kim Hazelwood [20]. They resorted to perf tool and hardware counters to track different hardware events at run-time. SPEC2006 INT benchmarks were run by them under *Pin* [3] and *DynamoRIO* [8] to capture their experimental data. Their data suggests that the cause of overhead was due to a spike in instruction execution compared to normal application run, which induced large number of iTLB misses and L1 instruction cache misses. Our findings match with their results.

Apart from hardware, it is important to understand the change in execution flow within DBT systems for different applications to know other sources of hold-

ups at run-time. Data on applicaiton run-time, basic block translation time, number of code cache exits and so forth have been gathered by us for that purpose. Based on the finding about the role of excess instructions in causing bottlenecks, the next place to look would be the source of those instructions. Because the application runs under the control of DBTs, through translated code, in addition to the application instructions the other instruction component would be the DynamoRIO instructions that are executed when control exits code cache (home of translated code). Our gathered data on exits indicates two leading factors: *Basic block translation, Indirect branches*.

Prior work has been done on indirect branches with SPIRE [17] to have a mechanism for hot indirect branches in place during translation. Here, a trampoline is used at the source pc to take control to code cache instead of exiting. That can enhance performance with reduced switching to DynamoRIO code and thereby the excess instructions. It also maintains code transparency when dealing with self-modifying code through a new code space the size of source where the trampolines are added, leaving the original intact.

When dealing with indirect branches, DynamoRIO uses JIT-based compilation with *software prediction*, which is different than the trampoline method used in SPIRE. With work done on SPIRE with indirect branch intensive benchmarks, results suggest that the trampoline method is better than *software based*. The approach of *software based* method is a table with mappings between source and target addresses in code cache. As source can't be know until execution of instruction in code cache, all source pcs are compared and matched target is reached. However, if no match found, then the bigger lookup with all source to target mappings is to be searched, which can be inefficient.

With basic block translation, well-known optimization technique is caching code, where translated code is preserved in memory for the duration of application execution. Additionally, appropriate translated code blocks are linked together in code cache, to prevent the exit and boost the performance. It is used in DynamoRIO as well and will be covered along with other optimization techniques in the next chapter 3.

Prior work has also been done on heuristics with static branch prediction and program profile analysis [24]. That work deals with static code outside the application run to look for patterns like loops, which rely on code that might already have been translated at run-time. Our work needs heuristics to predict a block to be translated for the very first time and so, can not utilize such techniques.

Chapter 3

Background

In this chapter, we will discuss DynamoRIO in a more detailed fashion. The following sections will cover its high-level design and different optimization techniques incorporated by its developers to minimize its run-time overhead. This chapter would give a better understanding of DynamoRIO and thereby our work, which is targeted at improving the performance of DynamoRIO even further.

3.1 Role of Dynamic Binary Translator(DBT)

DynamoRIO is a Dynamic Binary Translator(DBT), which performs run-time code emulation at its core. That in turn aids the application to exhibit behavior that is not part of its original design. In simple words, a DBT would take an application with a given set of features/capabilities and extend it to manifest enhanced/advanced set of features/capabilities. This widens the range of tasks that need to be done as part of application execution at run-time, in addition to its normal execution flow. For instance, the application can be run with enhanced security [16, 18], can be run on a platform other than the one it was originally

designed for (portability [2, 21, 22]), can do profiling [4, 19], which is essentially collecting some application specific data at run-time, as part of its study. Since the application is being transformed at run-time, the DBT deals with the application binary or the application executable file. To allow the DBT to induce the desired behavior in the application, it must be run in the controlled environment of the DBT. Because the targeted application behavior is achieved by transforming the application binary, the DBT first disassembles/maps the original binary to corresponding assembly level instructions. Then, the copy of the assembly instructions is manipulated, by injecting additional instructions, to carry out the desired task. This transformed copy of instructions is then executed by the DBT to get the final transformed application behavior. So, the original application binary stays intact and a transformed copy of its code is ultimately executed.

3.2 High-Level Design and Optimizations of DynamoRIO

As mentioned in the previous section, DynamoRIO translates the original application binary or source code to transformed or target code. The process of translating the source code, instruction by instruction, to the target code, is called interpretation. Because it is instruction by instruction, it can be a tedious process and would affect the run-time performance of DynamoRIO.

Developers of DynamoRIO have resorted to various optimization techniques to bring down the run-time significantly, compared to the standard interpretation process. These optimizations make-up the DynamoRIO system, within which the source code translation is performed. The following sub-sections briefly describe the optimizations and their place in the overall DynamoRIO design.

3.2.1 Basic Block Cache

To take advantage of the fact that parts of application/source code might be executed multiple times, the code is translated in blocks, called: *Basic Blocks*, and placed in a separate Basic Block Cache, to re-use it later during the application run. That way, application would have the required target code, saving the time to interpret/translate the source code. Each basic-block has a tag, which would be the starting address of the first instruction in the block of code. This tag would be the source pc. Correspondingly, the target pc would be the starting address of the first translated instruction, placed in the code cache. However, a translated block placed in basic-block code cache is called: Fragment. A hash-table is also in place to store the mapping between the source pc(spc)/original code to the target pc(tpc)/translated code. Figure 3.1 shows the execution flow of the DynamoRIO system on a high-level.

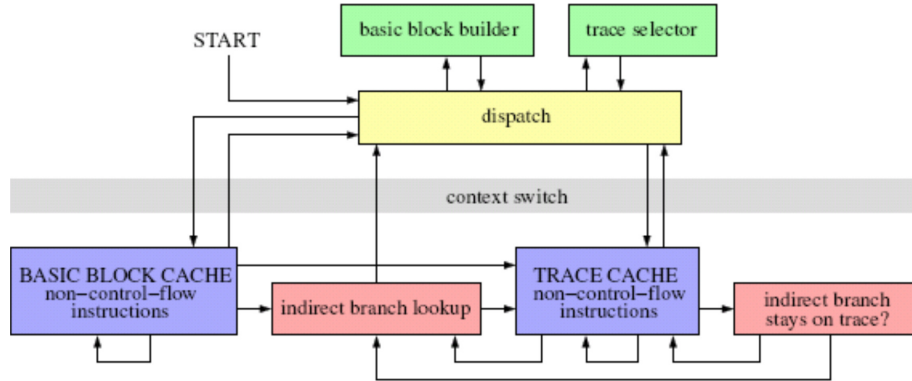


Figure 3.1. DynamoRIO execution flow from [11]

The *dispatch* is responsible for looking-up the hash-table to find a mapping for the spc to be executed next. If the corresponding tpc is found, the mapped fragment in code cache is executed. If a mapping is not found, then the basic block starting at the spc is interpreted/translated to create the required target

fragment, and is placed in the code cache. The new mapping between the spc and tpc is also added to the hash-table, to be found the next time the same spc is reached during execution. That would be the task of *basic block builder*, interacting with *dispatch*, as shown in below diagram.

The DynamoRIO has two contexts/modes in which it runs. The DynamoRIO context, in which the DynamoRIO code is run, to lookup the hash-table and to perform any required basic-block translation. If the required translated code is available in the code cache, then the context should be switched to get the execution control to code cache, where the translated fragment code can be executed. This would be part of the *Basic Block Cache*, shown in the control flow diagram.

3.2.2 Trace Cache

DynamoRIO does internal profiling to identify the basic block sequence executed quite frequently. A block of code is considered *hot* if its execution count reaches 50. It is marked as the *trace head*. The sequence of blocks after that constitute a *trace*, which represents a sequence of code blocks executed numerous times. As the length of a trace is finite, checks are in place to determine when to stop building the trace (terminating condition).

Apart from the basic block cache, DynamoRIO also has a trace cache(as shown in figure 3.1), to store the created traces during run-time. The idea behind caching traces is same as that of caching basic blocks. Traces are chains of code blocks known to be reached quite often, during the application execution. As there is a potential for those code sequences to be executed even more, translated sequence of code blocks are being cached as well. That way, the next time control reaches the trace head, wouldn't have to re-build the same trace. Like basic-block

mapping($spc \rightarrow tpc$), corresponding trace mapping is stored in a hash-table as well.

3.2.3 Linking Direct Branches

The caching of basic blocks and traces, avoids the need to repeatedly interpret each instruction, each time it is reached during application execution. However, research has shown that the main cause of DynamoRIO overhead is the execution of high volume of DynamoRIO instructions, when the execution control exits from the code cache. The control can exit the code cache for a variety of reasons, resulting in a context switch to DynamoRIO code. The reason for exit is then resolved by DynamoRIO code. With each exit, more and more DynamoRIO code is executed, causing the slow down. We will discuss different causes of DynamoRIO overhead in the next chapter. It would give an insight to the prior work done to investigate the bottlenecks. To prevent execution control from exiting the code cache, DynamoRIO links basic blocks with direct branches, if the translated code is already available in the cache. That way, the control can be kept within the code cache. As the target block is linked, the path to be taken in the execution is already known, without the need for DynamoRIO to resolve the next pc.

3.2.4 Indirect Branch Inlining

Another cause for control exiting the code cache is indirect branches. When executing an indirect branch, the target address/instruction can only be known when execution reaches that point. As an optimization, DynamoRIO places an indirect branch lookup table in the code cache itself, with different possible targets. The indirect branch lookup table is used by both basic block and trace caches, as shown in the flow diagram, to see if the target is already translated and available

in the respective cache's. A hit in the indirect branch lookup table, will ensure the control to stay within the code cache, preventing excess DynamoRIO instruction execution.

This chapter is mainly to showcase different optimization techniques employed by DynamoRIO developers to significantly bring down the application execution time, close to the native application run. However, research has shown that there is still a need to explore ways to minimize code cache exits and thereby preventing the high volume of DynamoRIO instructions from getting executed at run-time. As mentioned earlier, the run-time overhead prevents DBT's from being ubiquitous and there is often a trade-off involved with the use of DBT's and the resultant application run-time/response time. In the next chapter, we will discuss the prior work done to explore contributing factors to DynamoRIO overhead at run-time.

Our work primarily focuses on translating direct addresses, captured from the direct branches in different basic blocks, translated at run-time. The goal of our work is to further minimize the frequency at which the control escapes code cache, by availing the required translated blocks ahead of time within the code cache(analogous to pre-fetching code). That in turn would minimize the number of DynamoRIO instructions to be executed at run-time, as the linking would be in place, laying down the execution path for the application. Our approach banks on the direct branch linking technique, already incorporated in DynamoRIO. So, we are attempting to effectively extend that optimization technique, to potentially improve the application run-time within DynamoRIO.

In our pursuit of a solution, with the much evolved multi-core hardware, parallelizing the code translation process seemed like a plausible approach. We started-off by incorporating multiple compiler threads, such that each thread would be

responsible to translate a different basic block(or spc), picking one at a time from a pool of direct addresses. Like it was mentioned above, the collection of direct addresses is populated through all basic block translation's, done at run-time. This collection will be the input for all the compiler/translation threads to pre-compile target code, that can potentially be part of the guest application execution.

However, it has been found that the threads were lacking concurrency due to the string of locks implemented as part of DynamoRIO. We have explored ways to improve thread concurrency, but, there wasn't much success. A re-design of the DynamoRIO tool itself might be necessary to take advantage of thread parallelism. That would involve re-assessment of all the locks that are currently part of the code translation process, within DynamoRIO.

Some of our experimental results suggested that there is a lot of wasteful pre-compilation of basic blocks done at run-time. Thereby, the time spent by different threads making those translations is only exacerbating the problem of overhead. We then shifted our focus to finding a possible heuristics, tailored to predict the probability of a direct address or its basic block, to get executed next as part of program run. That way, only the most probable blocks to be executed next can be translated. With that strategy, any work done at run-time compiling target code will be relatively useful, which in turn can potentially improve the performance.

That was to give a brief description of our work, relating to the optimization technique: Linking Direct Branches, discussed in this chapter. A more precise and elaborate discussion of our work will be presented in following chapters, describing the challenges, different experiments, gathered results and analysis.

Chapter 4

Exploring Run-Time Bottlenecks

Prior research has been done on DBTs(Dynamic Binary Translators) in developing different optimization techniques to improve their performance. Those techniques have been discussed in the previous chapter 3(Background), within section 3.2. Even though DynamoRIO employs all those optimization techniques, there is still a considerable overhead when an application is run from within DynamoRIO. We did some preliminary work on DynamoRIO, to know the possible bottlenecks by investigating its source code. Numerous experiments have been run and results have been gathered to understand the run-time application translation and execution, under DynamoRIO.

4.1 Benchmarks and Environment set-up for experiments

Our prior work involved running experiments on a cluster platform constituted by Intel Xeon (R) W3530 2.8 GHz (x 8 cores) work-station's with 3.9 GB memory and 64-bit Fedora 18 operating system. The standard SPEC CPU2006 integer [7] and floating point [6] benchmarks have been used to capture different application

run-time statistics. Benchmarks are a standard set of programs implemented to test specific program features. They can be run with test or reference inputs. The test inputs run the benchmarks for a shorter duration and the reference inputs allow longer benchmark runs. The data gathered as part of our preliminary work will be discussed with respect to different benchmark test-inputs and reference-inputs.

4.2 Supporting Experimental Results and Analysis

Our initial experiments were targeted at gauging the performance overhead over a spectrum of applications. With some insight through the collected initial data, more specific experiments were designed and run, to dig deeper into the internal workings of DynamoRIO. The very first experiments were run to capture the application run-times and corresponding basic block compilation/translation times from within DynamoRIO. Those stats helped us see the extent of overhead in a variety of applications.

4.2.1 Overhead Measure: Run-times and Compilation-times

Different benchmark run-times and compilation-times(run under DynamoRIO) have been captured to know their relative performance, compared to their native runs. The ratio of DynamoRIO run-times to native run-times have been plotted in the Figure 4.1. If value of ratio is above 1, it represents overhead. A value below 1 means a speed-up, suggesting run-time better than the native run-time. The performance is same as that of native execution when the value is equal to 1.

As it can be seen in the graph of figure 4.1, benchmark runs involved two configurations: with traces(default) and without traces. The DynamoRIO run-

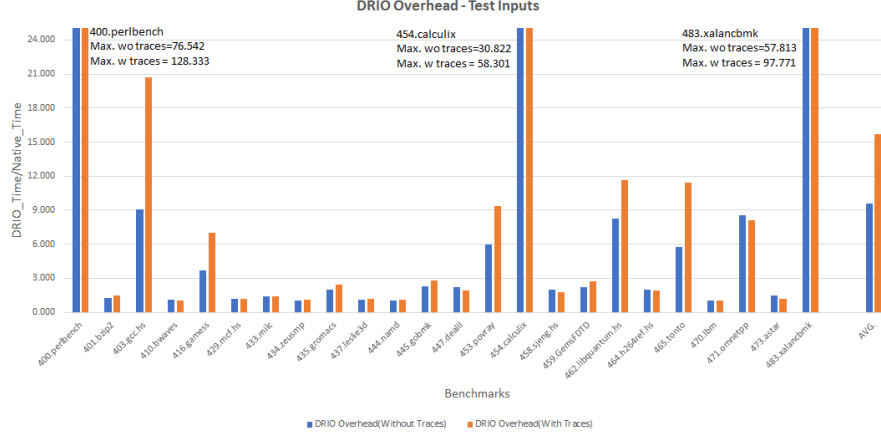


Figure 4.1. Ratio of *DynamoRIO* run times to *Native* run times for different benchmark *test inputs*

time option `-disable_traces` [9] has been used to capture benchmark run-times without traces. The overhead is high for benchmarks *400.perlbench*, *454.calculix*, *483.xalancbmk*. Out of those three benchmarks, *400.perlbench* has the highest overhead. Another inference from the graph is that on an average, the benchmark runs with traces took longer than the ones without traces. It suggests that traces as an optimization technique hasn't proven to be helpful for most of the benchmark runs with *test-inputs*.

The graph in figure 4.2 shows the *DynamoRIO* overhead for benchmark reference inputs, run with and without traces. The graph 4.2 has the plot of the *DynamoRIO* run-times to native run-times, similar to the one plotted for test-inputs 4.1. As it can be seen, on an average, the benchmark runs took longer without traces. Since the runs with reference inputs are longer, the traces as optimization technique helped in capitalizing on the code reusability aspect of the application execution. It thereby led to a relatively faster execution of the application, compared to the runs without traces.

The test-inputs with shorter run-times are more significantly impacted by the

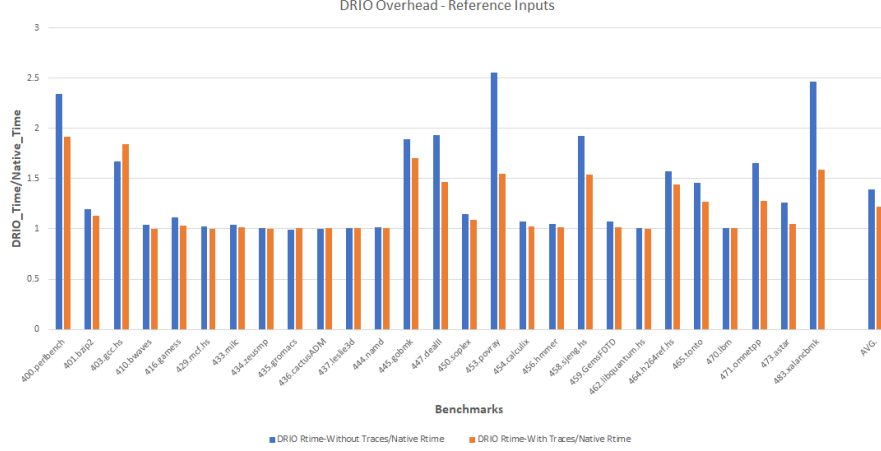


Figure 4.2. Ratio of *DynamoRIO* run times to *Native* run times for different benchmark *reference inputs*

DynamoRIO overhead. Due to their quicker program executions, the initial time spent in translating *basic-blocks/traces* for the first time, couldn't be compensated in the later part of the program execution with code reusability. Referring to the earlier inference made from the figure 4.1, with DynamoRIO overhead data for different benchmark test-inputs, the traces did not ameliorate their run-times. Based on that reasoning and the supporting experimental data, we decided to focus on test-inputs and to run further experiments without traces. So, all the following data is from experiments run without traces.

The benchmark compilation-times have been captured next. The figure 4.3 shows the clustered plot of three run-time numbers to give a high-level insight to the DynamoRIO overhead. Each benchmark has a cluster of three bars representing the flux in *total execution time*, *application execution time* and *basic block compilation time*, compared to the native run-times for different benchmarks. As it can be seen, more time is spent doing the basic block compilations, on an average.

Notable effect in performance is on the benchmarks: *483.xalanbmk*, *454.cal-*

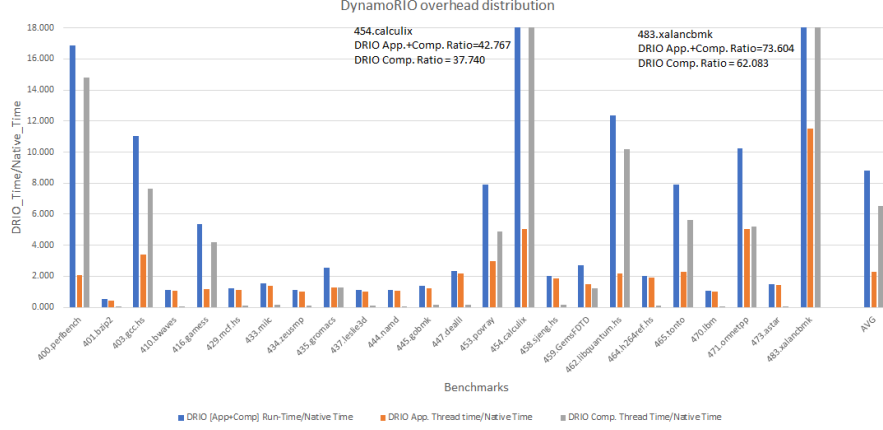


Figure 4.3. Run-Time overhead distribution: total/app. exec. time/compilation time, compared to native run-time.

culix, *400.perlbench*, where the compilation times compared to the native run-times are significantly higher. A relatively smaller, but, similar pattern can be seen in other benchmarks, like: *403.gcc.hs*, *416.gamess*, *453.povray*, *462.libquantum.hs* and *465.tonto*.

4.2.2 Code translations and Control Exiting Code-Cache

In an attempt to understand or trace back to the cause of high code compilations, additional data has been captured to know the number of entries to the dispatch module of DynamoRIO, exiting from the code cache, that resulted in basic block translations to target application code. The table 4.1 shows the total number of dispatch entries, number of basic-block translations and the additional entries to dispatch for reasons other than bb(basic-block) translations, for different benchmarks. The number on the third column is basically the difference of numbers in the first two columns. As the entries to dispatch happen only when control exits the code-cache, the non-zero number in the third column signifies the fact that the control is exiting code-cache to handle different contexts other

than bb translation.

Benchmark	Dispatch Entry Count	BB Trans- lation Count	Non-BB Dispatch Entries
400.perlbench	12145.429	9613.429	2532
401.bzip2	2629	2208.5	420.5
403.gcc.hs	67191	51934	15257
410.bwaves	4505	2957	1548
416.gamess	23583	10666	12917
429.mcf.hs	1784	1579	205
433.milc	22668	2945	19723
434.zeusmp	7341	6363	978
435.gromacs	6330	5303	1027
437.leslie3d	4994	4429	565
444.namd	5662	3485	2177
445.gobmk	9730.714	7582.571	2148.143
447.dealII	132322	12719	119603
453.povray	20170	10995	9175
454.calculix	11419	9582	1837
458.sjeng.hs	4605	3096	1509
459.GemsFDTD	12308	10227	2081
462.libquantum.hs	1671	1454	217
464.h264ref.hs	13657	6150	7507
465.tonto	25335	16563	8772
470.lbm	1293	1130	163
471.omnetpp	12347	7127	5220
473.astar	9321	2436	6885
483.xalancbmk	33871	24851	9020

Table 4.1. Different stats related to the benchmark test inputs

The information from DynamoRIO logs helped us see the distribution for different types of code-cache exits. The DynamoRIO run-time option `-loglevel 3` has been used to print the logs. The figure 4.4 portrays the distribution of code-cache exit types for different benchmarks. From the graph plot, it can be seen that the dominant exit reasons from code-cache are *Direct Target not in Cache*, *Indirect Branches*. The exit to handle the direct target address not in cache, is the case of missing target bb translation or code compilation leading to the afore mentioned run-time overhead in figure 4.3.

Additional experiments were done to track different parts of code in DynamoRIO, reached after the exit from code-cache. It involved the more expensive instruction stepping mechanism, where pre-determined points in code were marked as phases and then the number of instructions executed in different phases were recorded. The ptrace [10] tool was used to track the program progress instruction by instruction. That helped us see the exit reasons at a more finer granularity, apart from the need for code translations. Because of the single-stepping, the

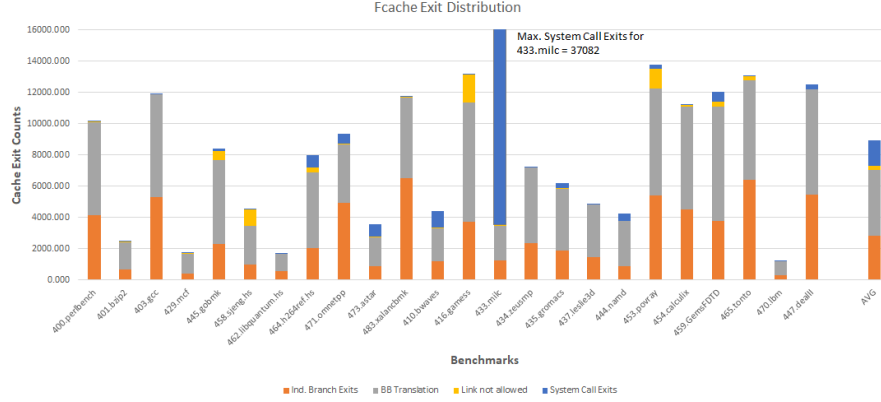


Figure 4.4. Distribution of fcache exit causes for different benchmark test inputs

benchmark runs were quite long.

The graphs in figures 4.5 and 4.6 show the recorded instruction counts in different DynamoRIO phases for benchmarks *456.hmmmer* and *400.perlbench*. Most of the executed instructions for benchmark *456.hmmmer* are in the code-cache (color: green), compared to the distribution for test-input *regmesg.pl* of benchmark *400.perlbench*, which is more random.

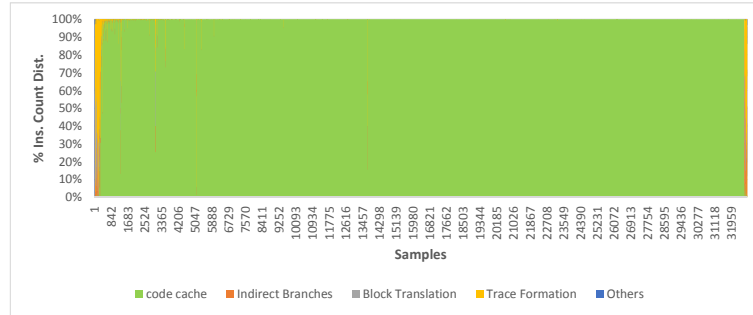


Figure 4.5. Distribution of instruction counts for different execution phases of *456.hmmmer* benchmark input bombesin

That can be related to the ratios of DynamoRIO to native benchmark run-times, plotted in the figure 4.1. The high overhead of benchmark *400.perlbench*'s test-input *regmesg.pl* can be explained by the inability of its execution control to

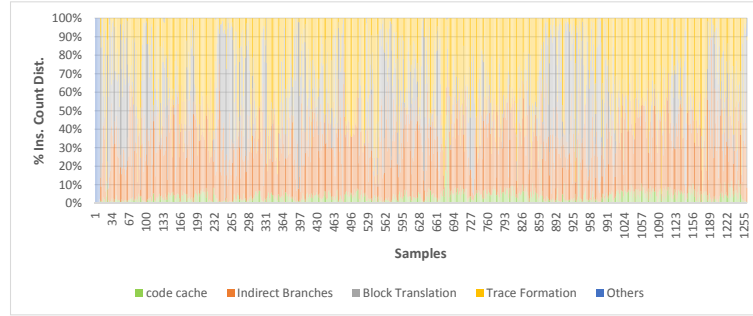


Figure 4.6. Distribution of instruction counts for different execution phases of 400.perlbench benchmark input regmesg.pl

stay in code-cache for longer intervals of time. Similarly, the lower overhead of benchmark *456.hmm* can be attributed to the more time-spent in code-cache.

The above findings laid the foundation for our work that will be discussed in the following chapters. The emphasis is on exploring ways to possibly keep control within code-cache, for extended intervals of program run. That is to prevent excess DynamoRIO instructions from being executed on each exit. Thereby, keeping the application execution time to the possible minimum. The different experiments carried out as part of our research will be described, along with the approaches that were tried out and analysis on the collected data.

Chapter 5

Eager Translation: Experimental Set-Up and Results

The basic block compilation in DynamoRIO is done in a lazy fashion, compiling one block at a time, only when the guest application needs the corresponding target code to continue with its execution. Our focus is to address the overhead associated with basic block translation. The idea is to translate basic blocks ahead of time, such that the required target code is ready to run when needed. For that to happen, we opted parallelization of the compilation process through multiple compiler threads. We call it eager translation, as the approach involves a shift from doing a lazy translation of basic blocks when needed, to translating all possible basic blocks prior to their need during application execution.

In order to generate multiple basic blocks through multiple threads at the same time, first we need to gather the known source pc's(spc's) which are the tags or the starting addresses of the blocks of code. Those known addresses are very much part of the basic block's themselves, as the target addresses of direct conditional/unconditional jumps. Below is a sample basic block:

```

0x00007fba1048ed6a  48 8b 93 08 01 00 00 mov    0x00000108(%rbx) -> %rdx
0x00007fba1048ed71  48 85 d2                test   %rdx %rdx
0x00007fba1048ed74  74 3d                  jz     $0x00007fba1048edb3

```

The last instruction is a conditional jump with accessible target address: *7fba1048edb3*. The design to implement our eager translation entails capturing direct addresses and populating them in a list. That list of addresses would be processed by multiple compiler threads ahead of time, with appropriate synchronization among all threads. It allows the required target code to be generated and stored in the basic block code cache, readily available for execution when needed.

The experiments discussed below with 1 or more compiler threads, have been run on the same 8-core cluster machines described in the preliminary work chapter 4.

5.1 Multiple compiler threads

The experiments have been run with thread count ranging from : 1 to 7. The maximum possible threads would be 8(max. 7 compiler threads + 1 application thread) to be mapped on to the 8 available cores. The eager translation with multiple compiler threads resulted in less translation requests from the application thread, but, did not help in improving the overall application execution time. Another experiment with eager translation was conducted with a static set of addresses reached during application execution. That involved disabling the address space layout randomization(ASLR) and running the application under DynamoRIO, to capture the required static addresses. With those known addresses, the applications/benchmarks were run with eager translation. Ideally, since the addresses being processed are certainly known to reach during applica-

tion execution, that should result in shorter run-times with more compiler threads processing more basic block translations at a given time. Even with that setting, the pattern of increased execution times was persistent.

The positive effect of eager translation in bringing down the translation requests with increased thread count, can be seen in the figure 5.1. Data in that graph has been collected from benchmark runs with eager translation of known static addresses. The seven clustered bars for each benchmark represent each of the seven thread configurations with counts: 1-7. The expected decreasing pattern in the number of translation requests can be seen in each cluster data plotted for each benchmark. For better visibility of the data pattern, plotted against the thread counts, the scale on the y-axis has been cut-down to 16000. The benchmarks *403.gcc.hs*, *483.xalancbmk* have counts ranging outside the 16000 mark and have been explicitly mentioned on the graph to have a view of the overall trend. The decreasing pattern of counts can be seen with increase of threads from 1 to 2 in the overshooting counts: *43144.800 to 24953.700* and *22702.100 to 16428.600*, for *403.gcc.hs* and *483.xalancbmk* respectively. On an average, the bb counts tend to approach a relatively steady pattern with more threads.

However, in some benchmarks the lowest counts can be seen for a lower thread count, rather than a steady drop in count with increase in number of threads. That can be attributed to the increased thread synchronization needed between the increased threads, causing the average basic block translation time to be greater with higher thread count. Another facet of the increased thread synchronzation impact, on different benchmarks, can be seen through the pattern of run-times in the figure 5.2. The increase in time with increased thread count can be seen in the consolidated time on an average. We found that the desired effect of faster

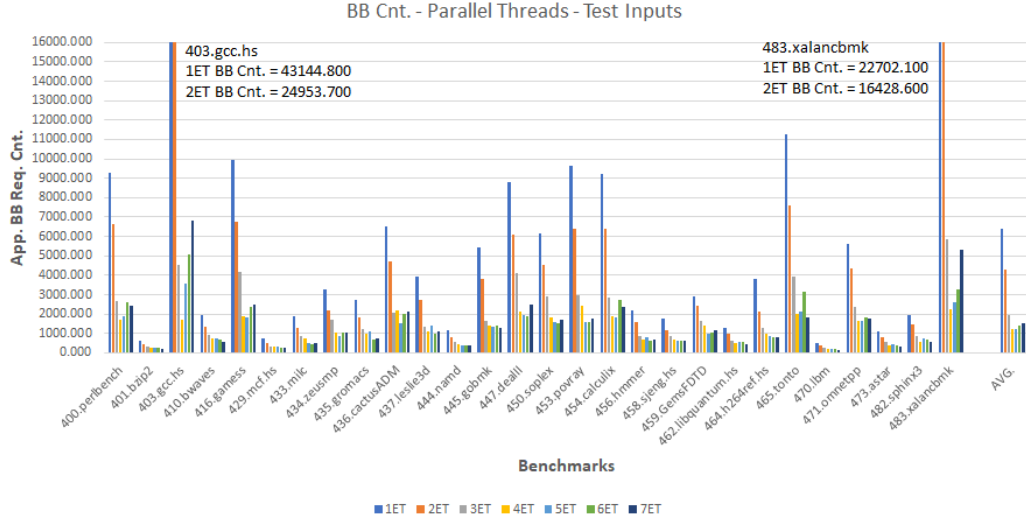


Figure 5.1. Basic block count captured with static list of addresses(*or without eager translation*), for different thread configurations.

benchmark execution couldn't be seen due to the string of locks in place at different stages of DynamoRIO execution of guest application. The increase in threads led to an increase in the lock contention, out weighing any performance improvement drawn from the increased bb translation. The next sub-section discusses another experimental setting and results, to showcase the impact of lock contention with more compiler threads.

5.2 Simulation of multiple compiler threads

This sub-section describes a simulation setting, where one compiler thread simulates the processing done by multiple compiler threads. It is based on a simple relation between the single compiler thread's processing time and the single application thread's run-time at any given time during an application run. Because it is a simulation of multiple compiler threads and have to account for the parallel processing of basic blocks, the single compiler thread is run for a dura-

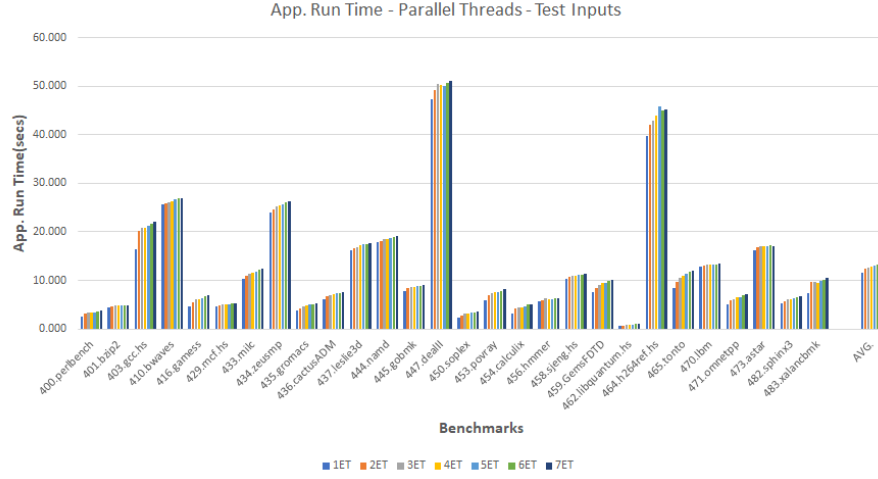


Figure 5.2. Application run-time with static list of addresses(*or without eager translation*), for different thread configurations.

tion of: *application thread run-time(at that point in program execution) * number of simulated compiler threads*. That way, we get the effect of more translation being done by the virtual threads being simulated with respect to the execution timeline.

Experiments have been run with and without eager translation(known static addresses captured with ASLR set to 0) to collect the application run-times for different benchmarks, with different thread configurations. Since it is a simulation of multiple compiler threads, the runs were done with simulated thread count set to even 10000. Accordingly, the time spent by the single compiler thread in translating the basic block was scaled-up in sync with the number of simulated threads. Unlike the increased run-time pattern, seen with the benchmark runs with multiple compiler threads(*figure 5.2*), the expected decrease in application execution time(*or a speed-up*) has been seen with the eager translation done with simulated threads.

The benchmark data has been captured through runs done with an overlap

of configurations, like: *with and without eager translation, thread counts: 1, 2, 4, 8, 16, 32, 64, 128, 10000(virtually infinite threads)*. That led to a bigger dataset. For that reason, it has been split in to multiple graphs to better showcase the data patterns. Separate graphs have been plotted for data with and without eager translation, and the data has been further broken down in to groups of thread configurations. The plot 5.3 shows data captured *without eager translation* for simulated thread counts: 1, 2, 4, 8. The decreasing pattern of run-times(*in seconds*) can be seen with increased simulated thread counts. The average values with the improved run-times for increasing thread count are also highlighted in a box on the graph.

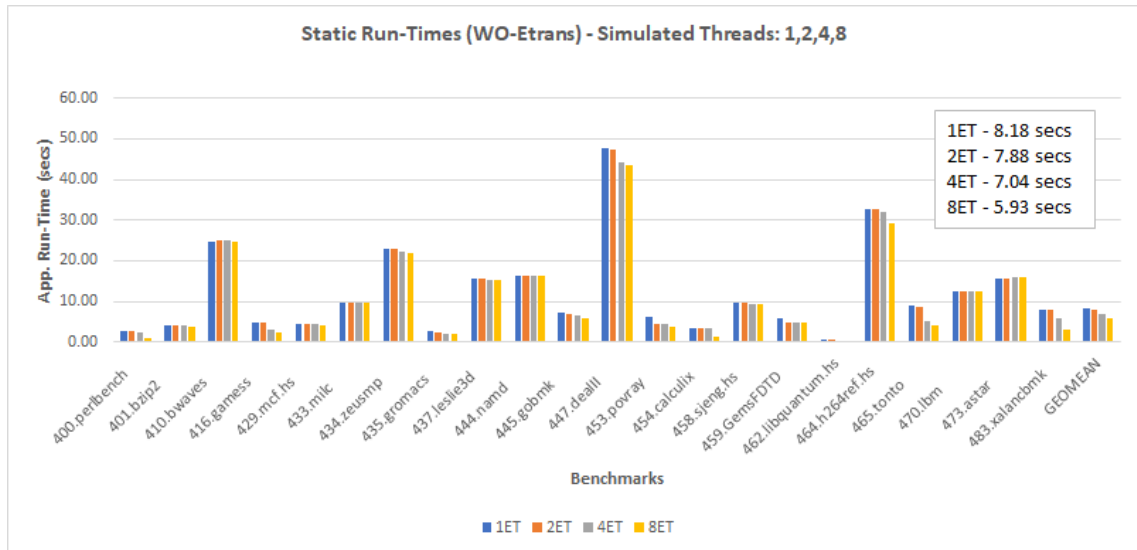


Figure 5.3. App Run-Time *without eager translation* for *simulated* compiler thread counts: 1,2,4,8.

Next is the graph 5.4, plotted with data captured *without eager translation* for thread configurations: 16, 32, 64, 128. The decreasing pattern or more accurately the transition to a steady-state can be seen in that graph for most of the benchmarks. However, there are exceptions like: *464.h264ref.hs*, where the best

times were for lower thread counts: 8, 16 respectively. That can be accounted to the excess translations that havenât aided in improving the application execution, but were still translated with increased simulated thread time. More discussion will be done on the usefulness of eager translation in the next chapter 6. On an average, the times tend to reach a steady state with increase in simulated thread count. The speed-up with *virtually infinite* simulated threads compared to the single thread has been found to be 56%, for this ideal case where only the code known to reach is translated. The relatively stable run-time pattern with increase in simulated thread counts can be seen with the average values inside the box.

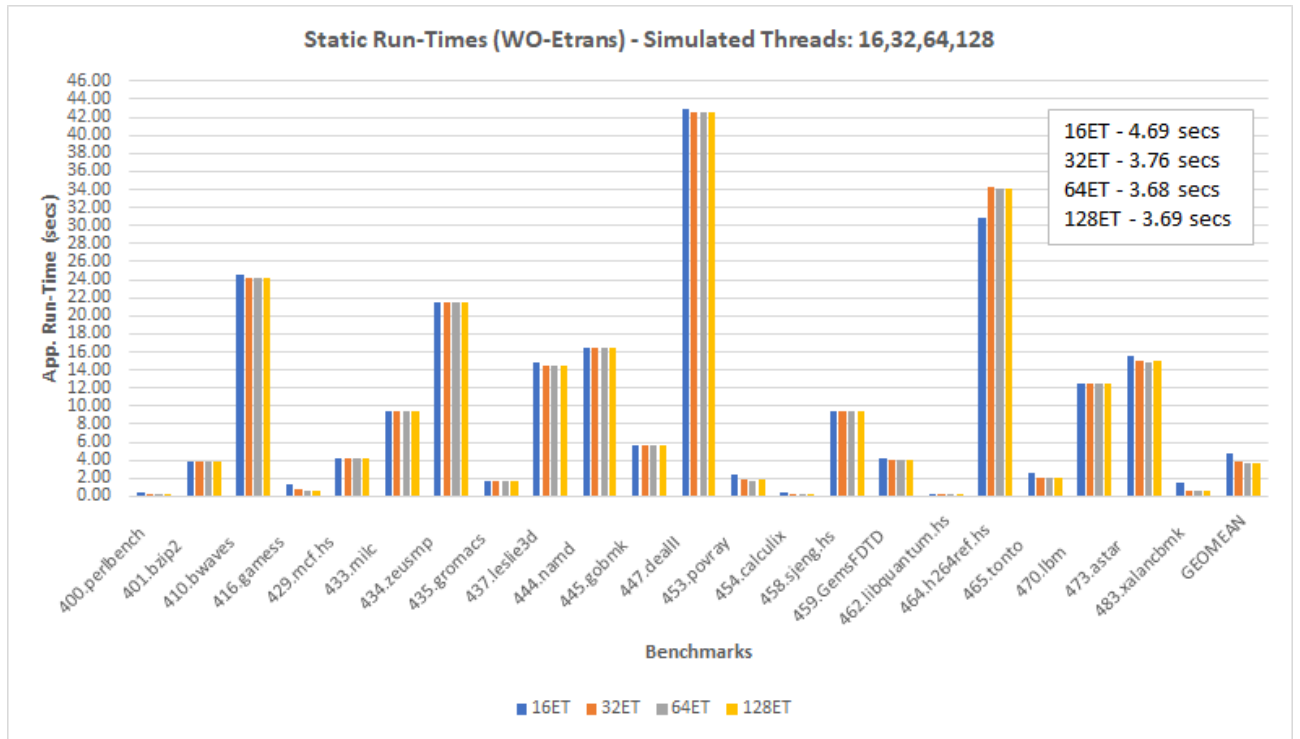


Figure 5.4. App Run-Time *without eager translation* for simulated compiler thread counts: 16,32,64,128.

The other set of graphs are *with eager translation*, with similar breakdown of thread configurations. The figure 5.5 shows the data for thread configurations: 1, 2, 4, 8. The decreasing time pattern with increasing thread counts can be seen in

most of the benchmarks and also on an average. A steady-state pattern can be seen for data plotted for thread configurations: *16, 32, 64, 128* in the graph 5.6, similar to the configuration run *without eager translation* in the plot 5.4. A speed-up of around *41%* could be seen with *virtually infinite* translation time, compared to single thread translation time.

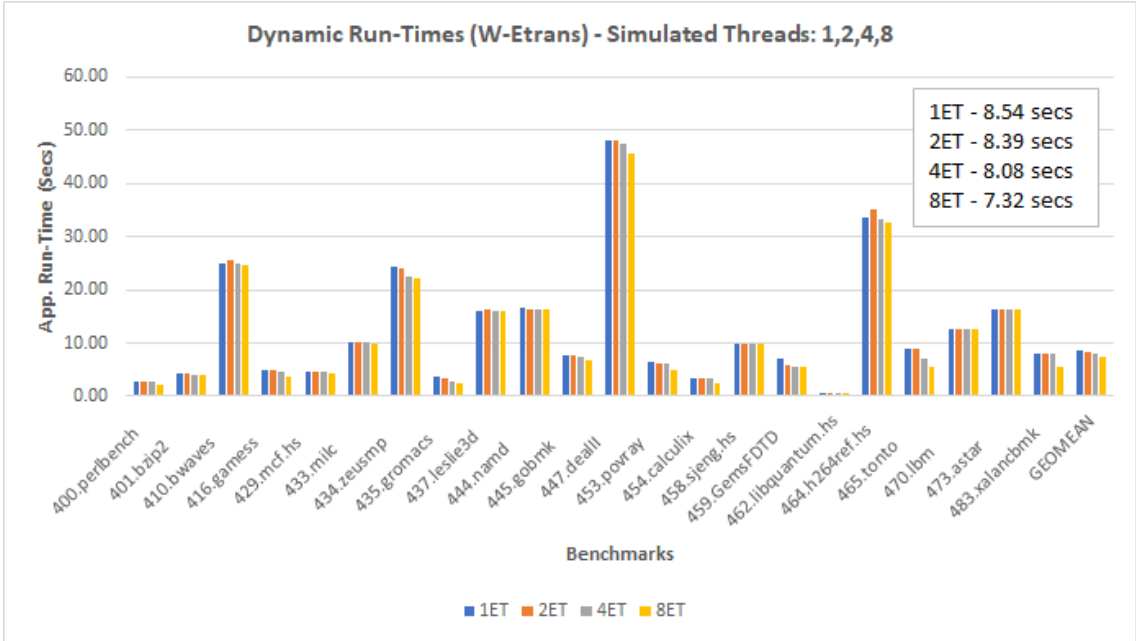


Figure 5.5. App Run-Time with *Eager Translation* for *simulated* compiler thread counts: *1,2,4,8*.

The simulation of 10000 compiler threads, or even 128, gives a lot of translation time for the single compiler thread to pretty much translate all possible basic blocks before the application thread can get started with its execution. That is the basis to classify the run-times attained with higher thread counts for both the configurations as a saturated state or the best possible run-time. The figure 5.7 shows the run-time comparison for data gathered with and without eager translation for the simulated thread count of 10000 (*virtually infinite threads*). The corresponding translation time given for the single compiler thread would then

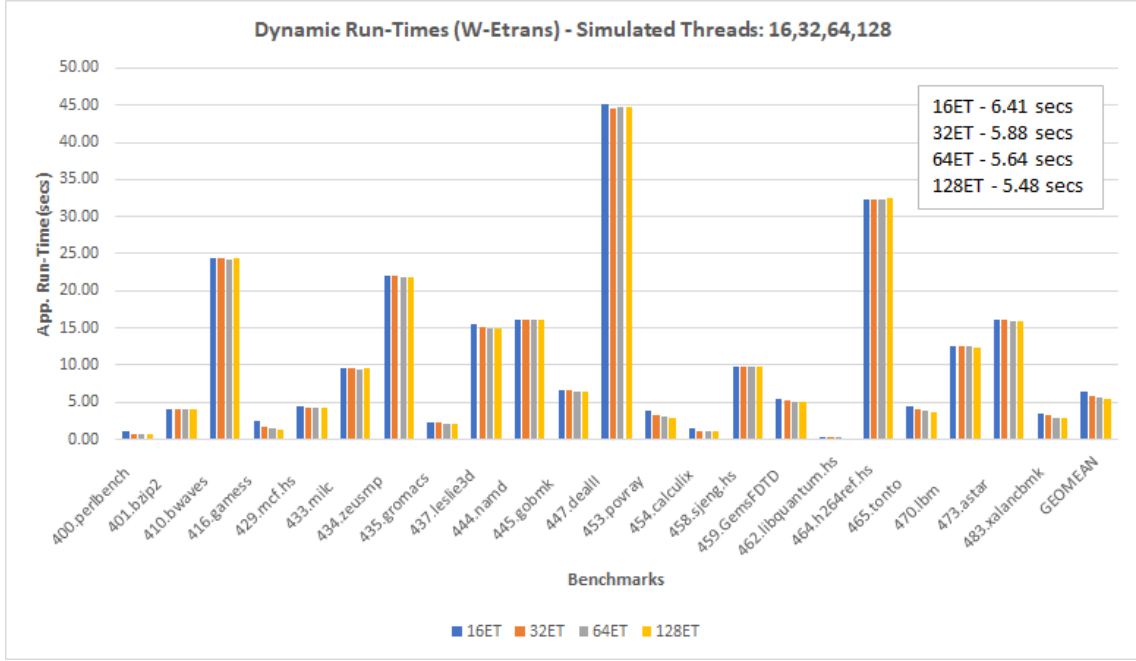


Figure 5.6. App Run-Time with *Eager Translation* for *simulated* compiler thread counts: 16,32,64,128.

be virtually infinite, with ample time to pretty much translate all possible basic blocks before the application thread can get started with its execution.

The graph 5.7 is a plot of run-time data ratios to know the performance *with eager translation* compared to *without eager translation*. A value below 1 indicates a speed-up or better performance and above 1 is overhead. Value of 1 is even performance in both configurations. As it can be seen, the configuration *with eager translation* has an average overhead of approximately 50% compared to *without eager translation*. There are many cases of even performance at value 1 and also a marginal speed-up of 3% for benchmarks *444.namd*, *464.h264ref.hs*. Note that the processed list of application/source addresses *without eager translation* are fixed set of addresses captured from prior application runs, and so are known to be reached for sure during execution. Where as, the addresses processed *with eager translation* are gathered at run-time during the current execution of the

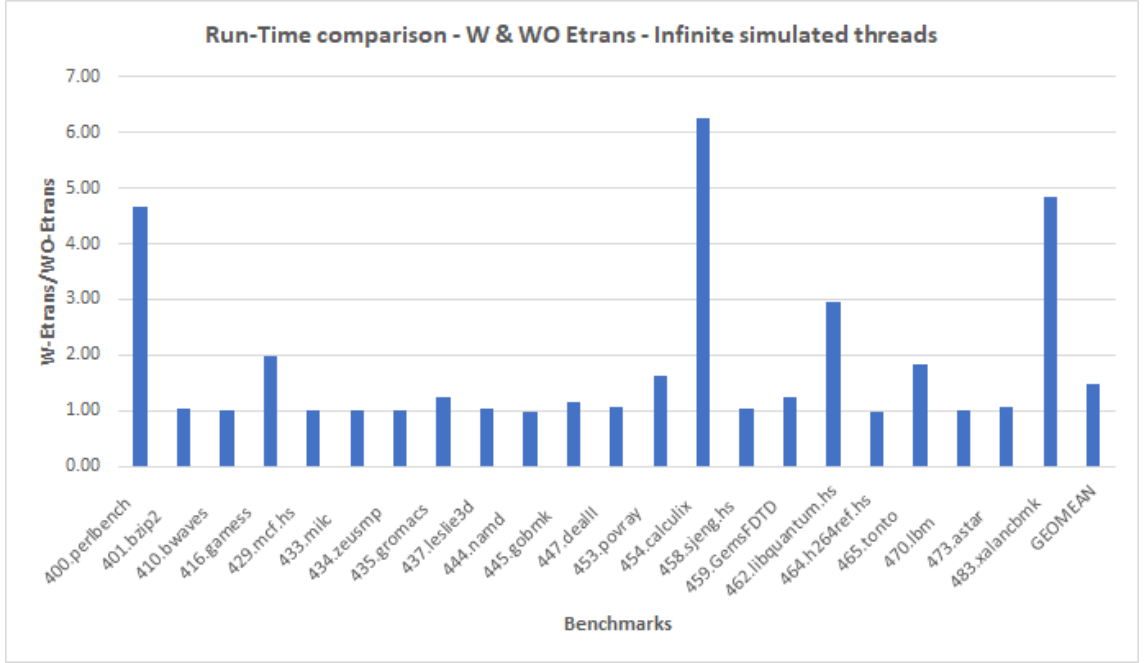


Figure 5.7. App Run-Time with and without *Eager Translation* for simulated compiler thread counts: *virtually infinite or 10000*.

application. For that reason, relatively more addresses might be translated in one go, within the translation time frame of 10000 simulated threads. Those translations are done irrespective of whether the block of code is reached during execution or not, unlike the configuration run without eager translation. Hence, the marginally higher run-times for benchmark runs *with eager translation*, due to the urgent request stalls from application thread.

Like it was mentioned in the previous sub-section 5.1, the difference between the multiple parallel threads and simulated multiple threads is the absence of the lock contention which normally gets manifested with multiple threads. This data from the simulated threads shows that the idea of eager translation can indeed result in an improved application performance. However, its realization through actual parallel threads would need a re-design of the DynamoRIO tool itself, with respect to the series of locks and their hierarchy, set through out its execution at

different stages (which essentially avoid deadlocks with multiple threads).

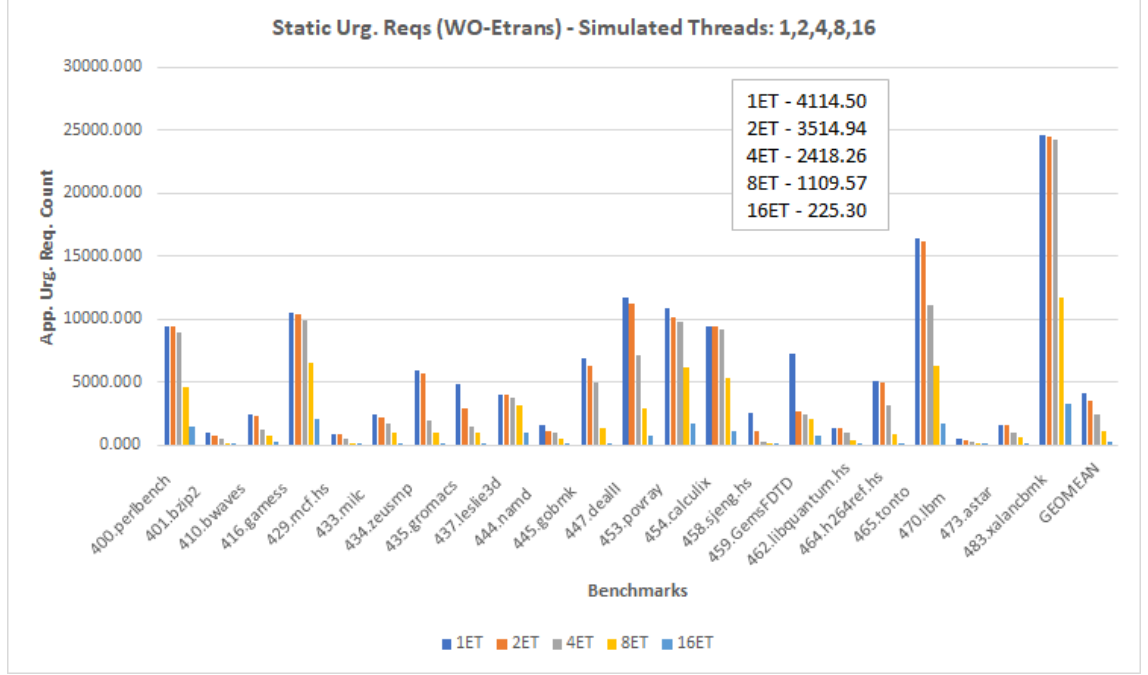


Figure 5.8. Requested BB translations without *Eager Translation* for *simulated* thread configurations: 1, 2, 4, 8, 16.

Next is the basic block request counts tracked in the application thread. Corresponding data has been broken down in to smaller data-sets, similar to the earlier data on run-times. The graphs 5.8 and 5.9 show the plot of basic block translations, requested by the application thread *without eager translation* for thread configurations: 1, 2, 4, 8, 16 and 32, 64, 128. The plots show a decreasing pattern of the counts, with increase in simulated thread count. For number of block requests for *virtually infinite* simulated thread count is only 0.07% compared to the single simulated thread configuration, which is the significantly high drop for ideal case. The pattern can also be seen with explicit average values highlighted in the graphs, within a box.

The other set of graphs *with eager translation* are 5.10 and 5.11 for thread

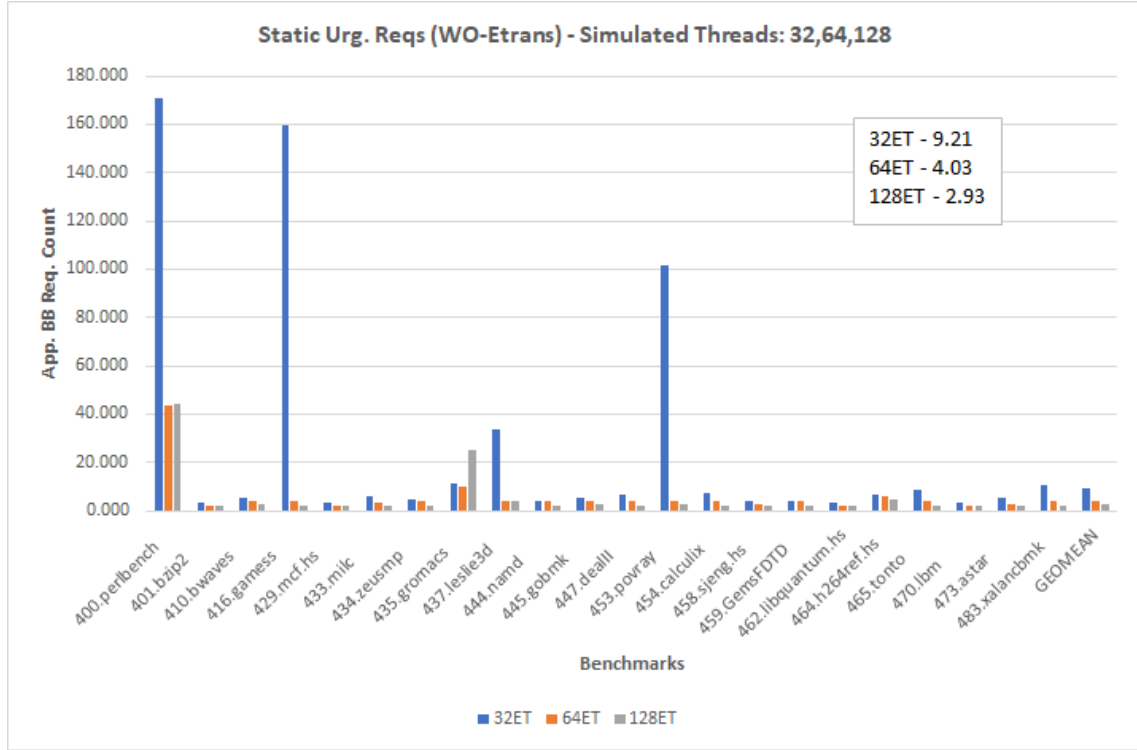


Figure 5.9. Requested BB translations without *Eager Translation* for *simulated* thread configurations: 32, 64, 128.

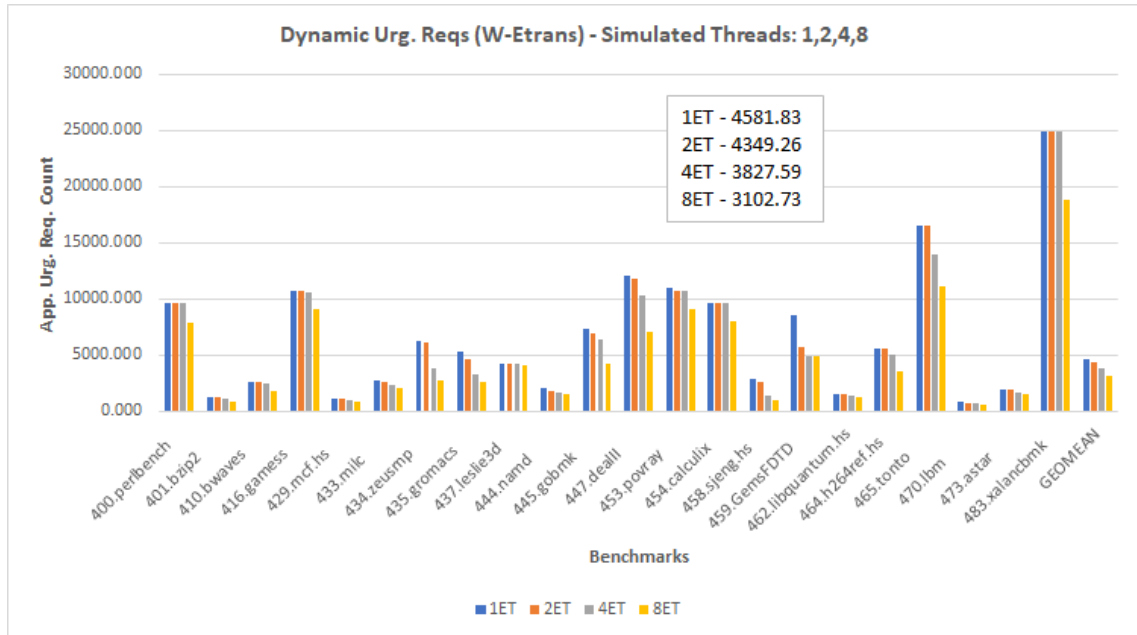


Figure 5.10. Requested BB translations with *Eager Translation* for *simulated* thread configurations: 1, 2, 4, 8.

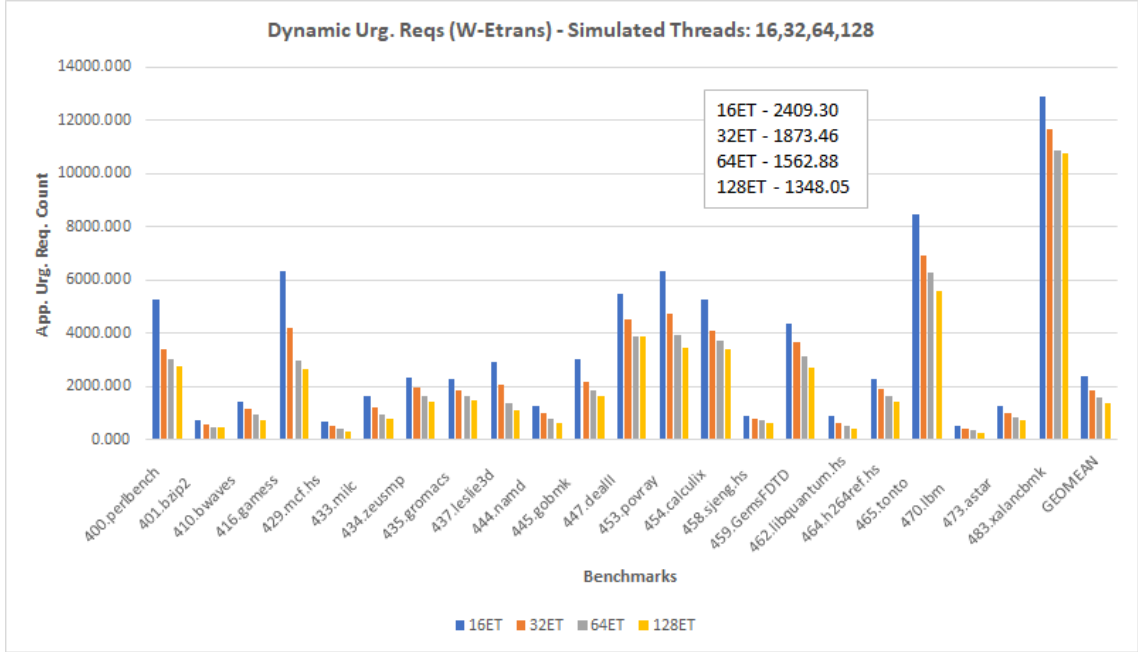


Figure 5.11. Requested BB translations with *Eager Translation* for simulated thread configurations: 16, 32, 64, 128.

configurations: 1, 2, 4, 8 and 16, 32, 64, 128 respectively, which also show a decreasing pattern of bb counts with increased thread counts. Similar pattern of basic block counts could also be seen with the prior setting of multiple compiler threads in the graph 5.1. On average the *virtually infinite* threads have 29% urgent requests/stalls compared to single thread.

The plot of data for virtually infinite threads is shown in graph 5.12, with and without eager translation. The graph has two y-axis's, for the two configurations. The lower bb count scale on the left-hand side represents the data *without eager translation*, and the higher bb count scale on the right-hand side represents the data *with eager translation*. It has been done to show a clear distinction between the range of counts logged for both the configurations. Since the 10000 simulated threads give virtually infinite eager translation time for the single compiler thread, those counts represent the lowest possible translation requests needed by

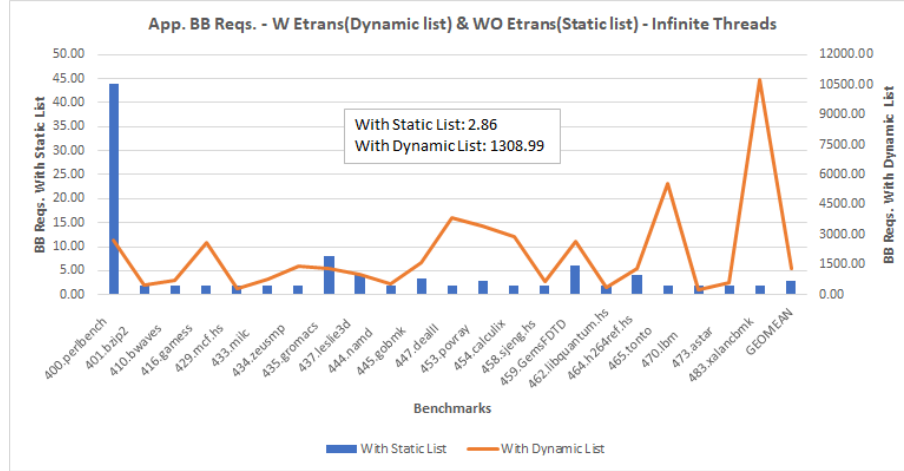


Figure 5.12. Requested BB translations with and without *Eager Translation* for different number of *simulated* compiler threads.

the application thread to successfully complete its run. Clearly, there are lot more translation requests from the application thread with runs involving eager translation (1308.99), than the runs without eager translation (2.86) and there is scope for improvement.

Because our eager translation strategy involves processing only the direct target addresses, the residual counts can be attributed to the cases like indirect branches, no linking between certain types of basic blocks. The 10000 simulated thread configuration is to gauge the extent of performance improvement that can be achieved by eagerly translating all possible blocks. However, since it is a simulation, those run-times and bb counts could be achieved by first doing all the translations and then running the readily available target code. In order to accomplish that with multiple compiler threads, apart from the hurdle of lock contention(discussed in section 5.1), another aspect of eager translation is to be addressed.

It is the percentage of eagerly translated basic blocks aiding in the application run by readily being available when needed. Based on the collected data, we found

that the percentage of useful eager translations is quite low. The next chapter discusses more about the results relevant to this aspect of eager translation and the need for a heuristic to better predict and translate basic blocks with higher probability of execution during the application run.

Chapter 6

Need for a BB Prediction Heuristic

Picking up on the discussion toward the end of previous chapter 5, only a smaller fraction of eagerly translated basic blocks are contributing to the enhancement of application execution. In other words, there is no filter on which basic blocks are to be translated from the pool of gathered addresses for eager translation. Depending on the input to the application, parts of the code base will be reached to process the same. So, direct target addresses for conditional branches may or may not be taken, based on the application state at that point in the program execution. All the direct addresses are gathered during the translation of all basic blocks, which serve as input for eager translation. Since all of them are translated irrespective of whether they are likely to be reached during the execution or not, more time is being spent doing unnecessary translations resulting in application thread pausing its execution due to the missing target translated code. Once the required block is translated and placed in the basic block code cache, the application would then be able to resume its execution through the

available target code. There is a need for a heuristic/algorithm that can predict the probability of likelihood for a basic block to be executed, possibly based on the prior translations.

Th figure 6.1 shows the clustered data of the fractional useful eager translation, for different *simulated* thread configurations, plotted for different benchmark test-inputs. The counts captured for runs with one simulated thread have been used as a baseline/threshold to gauge the usefulness of eager translation when run with higher simulated thread counts. The maximum fraction of eager translations that were readily available to boost the application run is 0.31 or 31% for the benchmarks: *483.xalancbmk* . It is not a big fraction and emphasizes the fact that a lot of unproductive work or translation is being done by the compiler thread. To get similar run-times with multiple threads, we need a way to predict the likelihood of a particular basic block to be executed as part of application run. The excess translation can then be kept to a minimum, thereby maximizing the run-time efficiency. With DynamoRIO, there is the additional problem of lock contention. The prediction algorithm/heuristic can then be used in other Dynamic Binary Translators(DBTs) or in a more generic setting.

To better understand the data and math involved in computing the percentages of useful translations, the table 6.1 shows the parameters and the benchmark data for the configuration with 10000 *simulated* threads. The second column shows the number of bb translations requested by the application thread when run with one simulated compiler thread. That is the baseline against which the counts captured for 10000 threads(in third column) are compared to quantify the usefulness of eager translation. The eager translation is minimum when done with one simulated compiler thread, as the translation time allotted for the actual

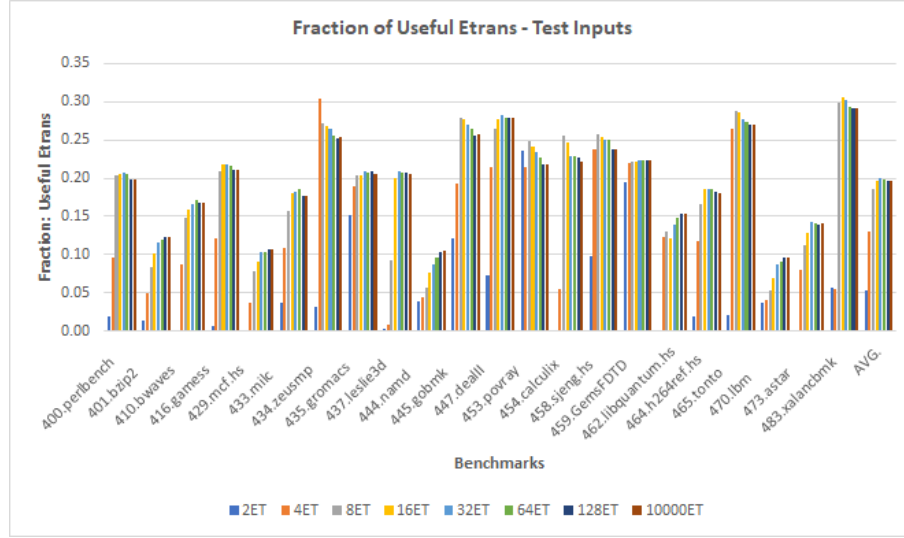


Figure 6.1. Fraction of useful eager translations for different benchmarks and different number of *simulated* compiler threads.

single compiler thread is also a minimum compared to the time given with higher thread counts. Hence, is the baseline/threshold to measure the effectiveness of eager translation with higher simulated thread count.

Benchmark	App. Req. Cnt.-1ET(Baseline)	App. Req. Cnt.-10000ET	Readily Available BB Cnt.	Etrans. Cnt. 10000ET	Fraction-Useful Etrans.
400.perlbench	9609.671	2731.343	6878.328	34745.843	0.20
401.bzip2	1299.35	457.1	842.25	6891.95	0.12
410.bwaves	2653	744.5	1908.5	11360.2	0.17
416.gamess	10659.3	2620	8039.3	38152	0.21
429.mcf.hs	1145	333.3	811.7	7604.5	0.11
433.milc	2671	784.8	1886.2	10621.6	0.18
434.zeusmp	6193.1	1455.4	4737.7	18679.2	0.25
435.gromacs	5241.9	1330.9	3911	19031.1	0.21
437.leslie3d	4224.4	1011	3213.4	15582	0.21
444.namd	2021.8	571.5	1450.3	13834.9	0.10
445.gobmk	7278.186	1617.757	5660.429	22074.471	0.26
447.deall	12039.6	3857	8182.6	29308	0.28
453.povray	10989.1	3431	7558.1	34570	0.22
454.calculix	9574.6	2902	6672.6	30095	0.22
458.sjeng.hs	2849	638.8	2210.2	9270.1	0.24
459.GemsFDTD	8470	2645.6	5824.4	26188.1	0.22
462.libquantum.hs	1447	390.4	1056.6	6849.6	0.15
464.h264ref.hs	5612	1334.9	4277.1	23704.7	0.18
465.tonto	16558.9	5561	10997.9	40786	0.27
470.lbm	797.6	261.2	536.4	5541.7	0.10
473.astar	1966.3	613.6	1352.7	9632.7	0.14
483.xalancbmk	24844.4	10749.7	14094.7	48441.4	0.29

Table 6.1. Computation of fraction useful eager translation for different benchmarks run with 10000 *simulated* threads.

With the increase in thread count, the translation time given to the single compiler thread proportionately increases. The increased translation places more

basic blocks in the code cache, resulting in a lesser translation requests from the application thread. So, the usefulness of eager translation is quantified by the decrease in the bb requests with increase in simulated thread count. Since the baseline(max. app. bb. requests) is the data associated with 1 simulated thread, the difference in counts between the 1 simulated thread(column 2) and 10000 simulated threads(column 3) gives the drop in the requests, captured in the column 4(*Readily Available BB Cnt.*).

The goal is to compute the fraction of eager translation that turned out to be useful to the application run. That drop in requests is caused by the eagerly translated blocks in the compiler thread. The total number of eagerly translated bb's for different benchmarks is in column 5(*Etrans. Cnt. 10000ET*). The ratio of numbers in column 4, which is the drop in requests, and column 5, which is the total eager translations, would give the fraction of the total eager translations that indeed led to an enhanced application execution. Those ratios have been included in column 6(*Fraction-Useful Etrans.*). The maximum fraction or percentage of usefulness for 10000 threads is for the benchmark *483.xalancbmk*, with value *0.29* or *29%* respectively. On an average the maximum is at *20%*.

To get similar run-times with multiple threads, we need a way to predict the likeliness of a particular basic block to be executed as part of application run. The excess bb translation can then be kept to a minimum, thereby maximizing the run-time efficiency. Work with heuristics or how they were generated and applied will be discussed in the next chapter 7, followed by experimental and data analysis in chapter 8.

Chapter 7

Generation and Application of Heuristics

7.1 Heuristics Through Data Mining

We opted to explore data-mining techniques to generate heuristics/rule-sets that could be applied at run-time during application run under DynamoRIO. Data-mining and machine-learning techniques are quite ubiquitous now with a range of libraries available to run different algorithms written in a variety of languages like Python, Java, etc. Big companies are also extensively investing in machine learning, like Google, Microsoft, Amazon through Tensorflow, Azure ML Studio, Amazon Machine Learning respectively. However, LEM1 [13] (Learning from Examples - Module 1) algorithm has been used for our work which is part of the LERS (Learning from examples using rough sets) data mining system.

The idea is to have a set of simple rules/heuristics as they would have to be applied at run-time without incurring much overhead. These rules will be applied

in conjunction with eager translation, to make a prediction on the likely basic blocks to be executed next which enables a more selective translation and a potential solution to reduce the excessive block translation at run-time. The LEM1 algorithm looks for patterns in the global space of all attribute values. Below is some terminology to understand the input data for the algorithm and the types of rules/heuristics. The description is less formal on purpose for easier understanding.

Data-set: Records of a combination of data attribute values leading up to a decision attribute value.

Concept: Set of data records with the same decision attribute value.

Partition: Set of groupings of records with same data attribute/decision attribute values. Represented with a "*" in the end.

Elementary-Set: Each group in a partition is called an elementary set.

Data-set consistency: A data-set is said to be consistent if each elementary set in the *partition* of data attributes is coherent with one decision attribute value. If one or more elementary sets have entries mapping to more than one decision attribute value, then the data-set is said to be in-consistent, as the available data can not be used to make a concrete decision represented by the decision attribute.

Conceptual Variables: Depending on a choice of decision attribute value, all records can be put in two groups: (i) Ones with chosen decision value (ii) Rest of the records

It would be a way to discover any hidden rules for the chosen decision attribute value.

Lower Approximation: If a data-set is in-consistent, then a union of elementary

sets leading to a sub-set of entries within a *concept*. That would give entries that are consistent with the chosen *concept* which in turn depends on the chosen *conceptual variable*. Because the data-set is in-consistent, the next best thing is a sub-set of records that are consistent. As the mapping of records to a single decision value is certain, the generated rules are known as *certain rules*.

Upper Approximation: Similar to *lower approximation*, this is for in-consistent data-sets and is the other extreme. Instead of a sub-set, a super-set of elementary sets is chosen such that the intersection of each elementary set and the concept is a non-empty set. So, it would include entries/records from elementary sets which have more than one decision value. The rules thereby generated may not help us in making a certain decision as their ingredients include records with other decision values too. For that reason, the rules are called *possible rules*. The combination of attribute values may or may not lead to the decision value set in the rule.

Below is a sample data-set in table 7.1 [14] with a bunch of data attributes followed by a single decision attribute:

Rec. Number	Wind	Humidity	Temperature	Trip
1	low	low	medium	yes
2	low	low	low	yes
3	low	medium	medium	yes
4	low	medium	high	yes
5	low	medium	high	maybe
6	medium	low	medium	maybe
7	medium	low	medium	maybe
8	medium	low	medium	no
9	high	high	high	no
10	medium	high	high	no

Table 7.1. Sample data-set to decide on a *Trip* based on *Wind*, *Humidity*, *Temperature*

The data-set in table 7.1 has 3 data attributes: *Wind*, *Humidity*, *Temperature*, and one decision attribute: *Trip*.

The unique decision attribute values are the *concepts*: *yes*, *maybe*, *no*.

The *partition* for data attributes = $\{\{1\}, \{2\}, \{3\}, \{4, 5\}, \{6, 7, 8\}, \{9\}, \{10\}\}$,
partition for decision attribute = $\{\{1, 2, 3, 4\}, \{5, 6, 7\}, \{8, 9, 10\}\}$

As discussed above, *partitions* are made of *elementary sets*, which are groups with same data/decision attribute values. The data-set is in-consistent because the elementary sets: $\{4,5\}$, $\{6,7,8\}$ are groups of records with more than one decision value: $\{\text{yes, maybe}\}$, $\{\text{maybe, maybe, no}\}$, respectively.

For rule-induction, new data-sets have to be created for conceptual variables of respective concepts, which is 3 in number for the sample data-set. So, data-sets with two sets of records would be created with one group of records mapped to a concept/decision value and another with rest of records. That would help with generation of rules with lower and upper approximations for different concepts. Those would be the certain and possible rule-sets.

7.2 Input Data for LEM1

The requirement for heuristics stems from the need to predict and make a more selective eager translation, with the intent to reduce the overhead resulting from useless translation. The task is to predict the start address of the next basic block that is most likely to get executed. To fit in the data-mining component

for the desired prediction, first we need to create training data for the LEM1 algorithm. Because the prediction is for basic blocks, data for appropriate basic block attributes along with control flow of next basic block had to be recorded first with preliminary benchmark/application runs.

Our work only deals with basic blocks that have conditional branches with direct addresses, where the target address can be directly accessed from the branch instruction. Exploration for viable basic block attributes started with the thought of pooling in random ones and then cutting down on the list. Flags on random instructions like "test", whether it is part of the basic block or not and if the target address or the fall through was reached next. The resultant data-set, which had like 2 data attributes and one decision attribute couldn't be processed by LEM1 to yield a rule-set.

On second thought, as conditional branch makes decision based on the contents of EFLAGS register [23], focus was turned to instructions that can affect eflags within a given basic block. So, that led to inclusion of attributes related to eflags and instructions affecting them, which can give us patterns on control flow decisions. That is essentially the branch prediction we are interested in.

Below is a sample set of data records generated and used as input for LEM1:

The table 7.2 has 11 data attributes related to a basic block with information on instruction operands, name, eflags, conditional branch mnemonics, range of jump and direction of jump. It has been split for better presentation in the document. Each of those records lead up to the decision on branch target prediction

rec. #	instrName	opType1	opType2	opType3	opType4	opType5
1	test	Register	Register	None	None	None
2	cmp	Register	Immediate	None	None	None
3	test	Register	Immediate	None	None	None
4	test	Register	Register	None	None	None
5	test	Register	Immediate	None	None	None
6	cmp	Register	Immediate	None	None	None
7	test	1byteMemOffset	Immediate	None	None	None
8	test	Register	Register	None	None	None
9	test	Register	Register	None	None	None

rec. #	insEFlags	dcbMnemonic	dcbTestFlags	dcbJmpRange	dcbJmpDir	isDCBTaken
1	oszpc	z_short	z	77	forward	yes
2	oszpc	nz	z	188	backward	yes
3	oszpc	nz	z	188	forward	no
4	oszpc	z	z	537	forward	yes
5	oszpc	nz	z	512	backward	no
6	oszpc	nz	z	579	backward	no
7	oszpc	nz	z	27140	backward	no
8	oszpc	z	z	269	forward	no
9	oszpc	nz	z	339	forward	no

Table 7.2. Sample records with *basic block* data attributes and *branch prediction* decision attribute: *isDCBTaken*

recorded in the decision attribute: *isDCBTaken* (last column in lower half), with values: *yes*, *no*. The integer values in the field: *dcbJmpDir* will be further converted to range of integers by the LEM1 algorithm using discretization [12] process.

Code changes were made within DynamoRIO to gather the required data attribute information for each basic block. Linking between basic blocks was disabled to know the next address reached and marking the actual result of the condition in the branch. Without that, the control might stay in code cache and will not be able to get the branch prediction information, which is part of the training data.

7.3 Applying Heuristics in DynamoRIO

Once the input training data was available from earlier benchmark runs, next thing to do was to generate the rule-sets. The LEM1 algorithm was used to do the same. It involves finding minimal rules, in the sense that the generated patterns should have least possible attribute and value pairs. This is done by dropping one attribute at a time from left to right to see if data-set consistency is still preserved with lesser attributes. If not, it is retained and the next one is tried until there are no more in the right. But, if none of the attributes can be dropped, the original is retained.

The rules for resulting consistent data-set will be formulated through the partitions of data attributes mapped to the respective concepts. Additional step to make the rules more minimal, would be to see if any of the attribute value pairs can be dropped again from left to right, to have a set of data records still matching the decision value set in the rule. Having a minimal allows in making the desired decision with least possible data.

Below are some sample rules:

```
(dcbMnemonic, b) & (opType1, 4byteMemOffset) -> (isDCBTaken, no)
(dcbMnemonic, b) & (opType1, 8byteRelAddress) -> (isDCBTaken, no)
(dcbMnemonic, b) & (opType2, None) -> (isDCBTaken, no)
(dcbMnemonic, b_short) & (instrName, bt) -> (isDCBTaken, no)
(dcbMnemonic, b_short) & (opType2, 4byteRelAddress) -> (isDCBTaken, no)
```

An example of an attribute-value pair would be: *(dcbMnemonic, b)*. It can be seen that the pairs on the left-hand side of "→" are tied together with "&",

which represents an "and". So, for a given data record if all the attribute values from a rule match then the decision value would be the one on the right-hand side of " \rightarrow ". Since we are interested in branch prediction the values would be: *yes*, *no*, if the input data record matches with one of the rules. This is the "testing" phase, where existing rules are applied to each basic block data records gathered during application run within DynamoRIO. Corresponding code has been implemented in DynamoRIO to apply the generated rule-sets for selective eager translation.

The application of rules is pretty straight-forward to first gather each field data during every basic block translation and then compare it with the input set of rules. If there is a match between basic block pattern and an input rule pattern, then based on the branch target prediction in the matched rule, either the conditional branch target alone (decision value: *yes*) or the fall-through address (decision value: *no*) is added to the list of target addresses. So, the overall number of addresses eagerly translated is relatively selective when there is a rule match with decision value: *yes* or *no*, potentially reducing the excess eager translation incurring part of the run-time overhead. Both the addresses are being added for mis-matches as more translation can still lead to more readily available blocks in code cache, than no translation at all, which can result in application halt with missing code cache block. Ultimately, reduction in run-time overhead would depend on whether the selective eager translation helps with the next block needed for application execution. Also, note that we are only dealing with a portion of basic blocks that have conditional branches with direct target addresses readily available in the instruction.

The input rule-sets would be read in by DynamoRIO from files and loaded into a sorted array for an efficient binary search. The comparison of patterns is quite straight-forward. First the basic block pattern is gathered during translation and then compared with rules in the array through binary search. So, addresses would be added based on the first hit in the rule-set. More sophisticated techniques can certainly be used for better classification at run-time, but, might need more computations. Exploration of other algorithms and techniques is left for future work.

7.3.1 In-Domain vs Out-Domain Heuristics

When applying heuristics, we needed to know how well the rule-sets from training data could be applied/tested with random applications. Ideally, a standardized classification system for programs and their inputs would enable a more extensive use of rule-sets. If a rule-set is generated with a particular program and input, it could then be used with other instances of the same class of program/input or even a different class, based on the available classification system and needs of application. However, we currently do not have such a system at disposal. It needs exploration and could be another PhD material in itself.

So, the next best thing is a straight-forward logical classification. We have two broader domains: *in-domain*, *out-domain*. The domain names are quite intuitive and as they suggest, *in-domain* refers to the setting where a given application rule-set is applied to another application instance within the same domain. On the other hand, *out-domain* refers to the setting where a given application rule-set is applied to instances outside its own domain. These two settings are to know

how well a given rule-set can work for a variety of applications that are inside or outside a defined logical domain.

We have more finer granularity within those two domains, written below:

in-domain:

- (i) Rule-set generated for one application applied during its own run (more of a verification).
- (ii) Multi-input benchmark rule-sets.
- (iii) Logical domains *SPEC_INT*, *SPEC_FLOAT* - Inner Random Rule-Sets. For this configuration, rule-set of one *SPEC_INT* or *SPEC_FLOAT* benchmark is run against others within its respective domain.

out-domain:

- (i) Test input rule-set applied to reference-input runs.
- (ii) Logical domains *SPEC_INT*, *SPEC_FLOAT* - Outer Random Rule-Sets. Rule-sets of one domain applied for benchmark runs in another domain. This is quite an extensive set of stats to give us an insight on how well a rule-set can work with random applications.

Gathered data for these configurations will be discussed in the next chapter [8].

Chapter 8

Eager Translation with Heuristics: Experimental Set-Up and Results

This chapter is the follow-up of prior chapter[7] to discuss things like set-up within DynamoRIO to generate and apply heuristics. Then, different performance stats gathered for SPEC 2006 benchmarks by applying different rules/heuristics is analyzed. We start with the experimental set-up and then get to the data analysis.

8.1 Experimental Set-Up

The following sub-sections will discuss various configurations within DynamoRIO (as our research platform) to extract the required input data that LEM1 algorithm needs for rule-induction.

8.1.1 Extract Basic Block Input data for LEM1

First step in data-mining/machine-learning is to train a model. For that, need training data where the data-attributes, decision-attribute and their values are known through an expert or other appropriate sources. Sample set of training data used in our work for the basic block branch prediction has been made available in table[7.2] of chapter[7].

It can be seen in table[7.2] that we ended up with 12 attributes (11 data-attributes, 1 decision-attribute: *isDCBTaken* for branch prediction). Each of the data record in that table is from a single basic block, extracted during its translation within DynamoRIO. As mentioned in previous chapter[7], the focus was turned to instructions within a basic block that affect eflags, as contents of eflags register would be used by the conditional branch instruction to make the decision on whether the target address or the fall-through (address right after the conditional branch instruction) should be reached next. To incorporate that in DynamoRIO, and deal with instruction-level data, mappings between instruction names, opcodes, mnemonics (for conditional jump instructions only), eflags affected by those instructions was made part of the code. That enabled look-up for each instruction at run-time and gathered information only from instructions that were flagged as per our set profile. The new data-mappings in DynamoRIO for this setting are different arrays of structs modeling the mappings. Arrays are also ordered for an efficient binary-search. There might be more than one instruction affecting eflags in a basic block, before control reaches the

conditional branch instruction in the end. The data collected is only for the last instruction affecting eflags. So, had a data structure implemented in DynamoRIO that could track and change information with each new instruction.

Accordingly, the attributes are data on instruction name (ex: test, cmp, add), source/destination operands (type: Register, Immediate, so forth), eflags affected by the instruction (like "o" for overflow), along with the conditional branch attributes like the mnemonic (ex: z for jz branch), range of jump based on target address with reference to current instruction address and also the direction of jump (forward or backward). All of this input data leading to a pattern for the conditional branch prediction could say whether the branch target address would be reached (prediction: yes), or not (prediction: no).

To capture the detail on address reached right after the current basic block (and thereby setting an appropriate prediction value), linking between basic blocks had to be disabled in code-cache. If not done so, the connected basic block would directly get executed in code-cache without control reaching DynamoRIO, where we have all our code to capture the intended data. The data would also be inaccurate as the address captured could be a random first exit from code-cache and not necessarily the one next to the current basic-block.

Data record for each basic block was then written to a file to be used later for rule-induction with LEM1.

8.1.2 Applying Heuristics During Eager Translation

With LEM1 algorithm certain (lower-approximation) and possible (upper-approximation) rule-sets could be generated, as the original data-sets generated through the process described in above sub-section 8.1.1 were inconsistent. Further more, we created a set of rules that were common to both certain and possible rule-sets, outside LEM1. These three rule-sets were used in different benchmark run configurations for gathering performance stats.

Changes in DynamoRIO included logic to read the input rule-set files, store them in an array for comparison with basic block patterns found during run-time translations. Since the comparison would involve data attributes discussed in sub-section 8.1.1, same strategy was applied here as well to gather that information for each basic block (bb). After the data was available, current bb pattern was compared with the rule-patterns read from the input file until the first hit. Depending on the prediction set in the first matched rule, if it was a "yes", then the conditional branch target was added to the list of addresses processed through eager translation. If it was a "no", the fall-through address was added. So, we are making the address list entries more selective through the use of rules/heuristics for eager translation, in an attempt to bring down the excess eager translation problem found in the earlier strategy, where both target and fall-through addresses were always added. However, if there is no rule-match, then both the addresses are added like in the past, as we still do not want many stalls in application execution due to missing translated code in the code-cache. Effectively, we reduce the needed *eager translations* by one when there is a match, as we only add either the target or fall-through address.

We then record numbers like, the number of urgent requests (stalls) from application thread during execution, number of eager translations processed by the compiler thread (excluding the urgent requests) and run-time for the application thread (in seconds). Additional stats are derived from these numbers to see if the usage of rules/heuristics with eager translation is of use. Data gathered in different run configurations will be discussed next through these stats.

8.1.2.1 In-Domain vs Out-Domain

The logical domains discussed in sub-section[7.3.1] of chapter[7] are necessarily the different run configurations for all the SPEC 2006 benchmarks. As mentioned in the previous chapter, these domains are to explore how effective heuristics can be with eager translation at application run-time. Details of each configuration will be discussed in the following sections.

8.2 Analysis of Experimental data

All the data has been collected with a simulated thread (*10,000 threads*) setting, providing virtually infinite time for the compiler thread to do the eager translation. Data for various run configurations is analyzed in specific sections below. Since this is an exploration with respect to how well heuristics fit in with eager translation, benchmarks are run with rule-sets generated for respective and other benchmarks (through LEM1 algorithm) and data was gathered for,

(i) Urgent Request Counts : Application thread stalls awaiting translated code

in code cache.

(ii) Explicit Eager Translation Counts : Number of basic blocks eagerly translated by compiler thread, excluding the urgent requests from the application thread.

(iii) Application Thread Time (*in seconds*) : Application thread time on cpu, extracted from the proc file-system (maintained by the operating system).

That data was in turn used to compile relevant stats to gauge the effectiveness/impact of heuristics on various benchmark runs and also the compatibility of each benchmark with random rule-sets. Below are the relevant stats:

% Useful Eager Translation: All the benchmark runs under DynamoRIO involved basic block translation done through a simulated thread configuration. Depending on the setting for simulated thread count (say 10,000 threads) and the application thread time at that time (say T seconds), compiler thread time was accordingly scaled-up ($10,000 * T$ seconds). So, the application thread and compiler thread run in a lock-step fashion, where eager translation in compiler thread was triggered by a stall in application thread due to a missing translated basic block in code cache. However, the list of addresses for eager translation was quite extensive as every possible basic block was being translated leading to excessive translation and code within the code cache. This excessive translation and thereby the extra code in code cache, results in a memory requirement

spike. It could be a limitation to the application of eager translation technique and may not be encouraging to label it as an optimization technique within a DBT system.

The introduction of heuristics is to have a more selective pool of addresses for eager translation, ameliorating the memory overhead with reasonable run-time performance. A way to quantify the usefulness of eager translation, is the drop in *urgent requests* from the application thread (compared to the urgent requests in a normal run without eager translation) against the *explicit eager translation count*. That is to know how much of the eager translation contributed to a relatively smoother application run by reducing the urgent requests (less application execution stalls), compared to a normal application run under default DynamoRIO without any eager translation. The goal is to have an increased percentage of useful eager translation. So, higher the better.

% Urgent Requests: This stat attributes to the number of urgent requests for runs with eager translation and heuristics, compared to the normal run. Because more translations would be done with eager translation ahead of time, the urgent requests should be less than those with normal run, where translation is only done as needed on demand. Low percentage of urgent requests is desirable at run-time as that would mean fewer application/benchmark stalls, enabling a more fluid execution.

The coupling of heuristics to eager translation would lead to relatively fewer

translations. That might make way to an increased percentage of urgent requests, compared to the run configuration with eager translation alone, without any heuristics. As the goal of heuristics is to address the huge volume of run-time translation/memory overhead, reduced translations (or reduced memory requirements) would be a trade-off in terms of application run-time performance with possibly increased urgent requests (execution stalls). More specific analysis on data patterns will be discussed in the following sections.

Run-Time Overhead: This is the application thread run-time overhead to know the impact of eager translation and heuristics. It is the ratio of application thread time (with eager translation+ heuristics) to the normal application thread time (with no eager translation and no heuristics). Both the application runs are under DynamoRIO and the normal run is the baseline stat against which other run configuration stats are measured/quantified. Ratio below 1 means a speed-up compared to normal run and anything above 1 is an overhead. Additional details in the upcoming sections.

All the tables presented in the following sub-sections have been partitioned with data in upper and lower halves for a better fit, given the large magnitude of data.

8.2.1 In-Domain

(i)Rule-set generated for one application applied during its own run (more of a verification):

In this configuration, the rule-sets generated for each of the SPEC 2006 benchmarks are applied during their respective runs. Given that context, this

configuration is more of a verification with the expectation to see some positive results as rule-sets are specifically tailored to each of the benchmarks. The stats discussed above have been plotted with data gathered for *certain*, *possible*, *common* (in *certain* and *possible*) rule-sets in the figures 8.1, 8.2 and 8.3 respectively.

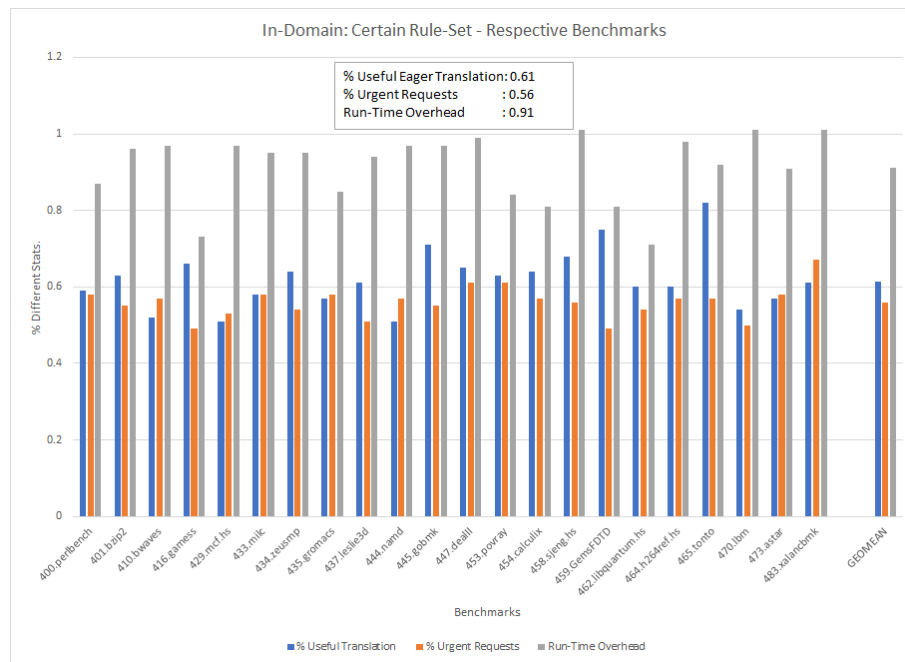


Figure 8.1. Comparison of run-time stats for the test input runs, applying *certain* rule-sets with *eager translation*.

With the certain rule-set data plot in figure 8.1 it can be seen that on an average (geomean), % *Useful Eager Translation* is at 61% with % *Urgent Requests* at 56%. The % *Useful Eager Translation* is significantly better than the configuration with excessive eager translation without heuristics. The selective eager translation did result in a higher percentage of urgent requests, but, could still get a speed-up of 9% as indicated by the *run-time overhead* stat marked at 91%.

The vision or expectation we had for heuristics/rules is that they would help in making accurate predictions on which addresses would be reached next, reducing the pool of addresses to be eagerly translated (less memory footprint) and making the application execution more continuous by reduced urgent requests if predictions were accurate. The rise in *% urgent requests* is highlighting the fact that the selectiveness in eager translation through heuristics is not of high accuracy and thereby an indicator of their quality. That in turn resulted in more breaks during application run. However, a decent speed-up of 9% could still be managed and contrary to the expectation there is an improvement in *% useful translation*.

Because the heuristics reduce the length of address list to be translated that would mean less memory for translated code in code cache (part of DynamoRIO system). So, even if the quality of rules/heuristics is not stellar and branch predictions are not that accurate (bump in urgent requests), element of selectiveness can still reduce the overall translated code and memory needs. The spike in *% useful translation* is due to that drop in overall eager translations along with a reduction in urgent request drop (compared to the requests in normal run without eager translation), not exactly due to the accuracy of branch prediction with heuristics. If reasonable speed-up can still be achieved with that setting, it could be open to applications on platforms with memory constraints. More data analysis on the drop in explicit eager translations with selectiveness will be covered later in this chapter.

With the awareness of low quality branch prediction, the choice between

applying heuristics for selective eager translation (lesser eager translations) in compiler thread that saves memory and the increased urgent requests in application thread that adds run-time overhead is more of a trade-off, which seemed to have worked well for certain rule-set, resulting in an overall application thread speed-up (dominant pattern with some outliers) with the benefit of lesser memory footprint.

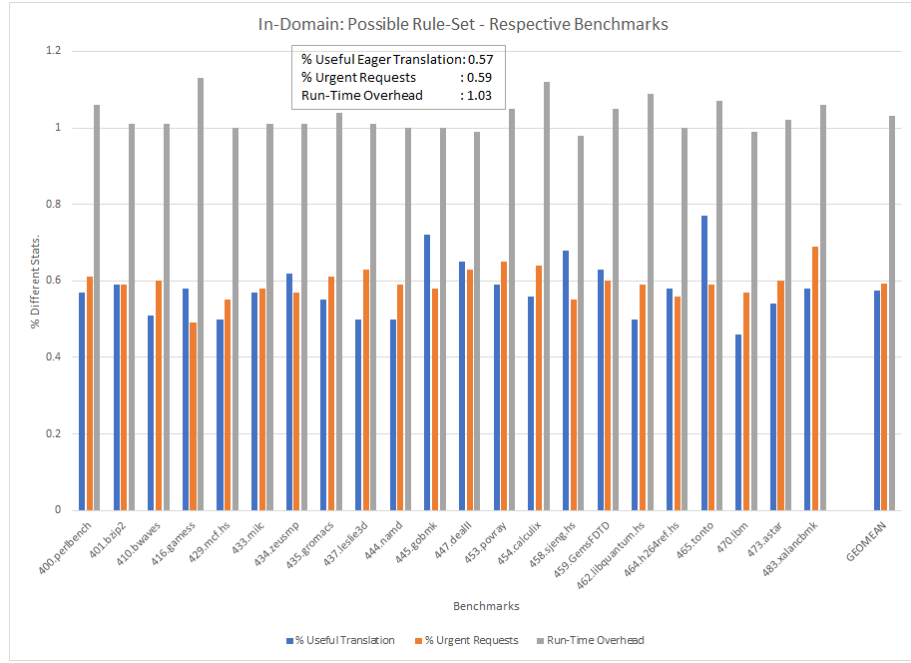


Figure 8.2. Comparison of run-time stats for the test input runs, applying *possible* rule-sets with *eager translation*.

With the data plot for *possible* rule-set in figure 8.2, the two percentages are quite close with scales tipping little toward the opposite end of pattern found in *certain* rule-set data. It can also be seen that *run-time overhead* > 1 in most of the cases. So, for the *possible* rule-set, the trade-off in selective eager translation did not work that well with the increased urgent requests and led to an overhead in most cases. It could still be an application case for constrained memory, if it

is an acceptable overhead for a given context.

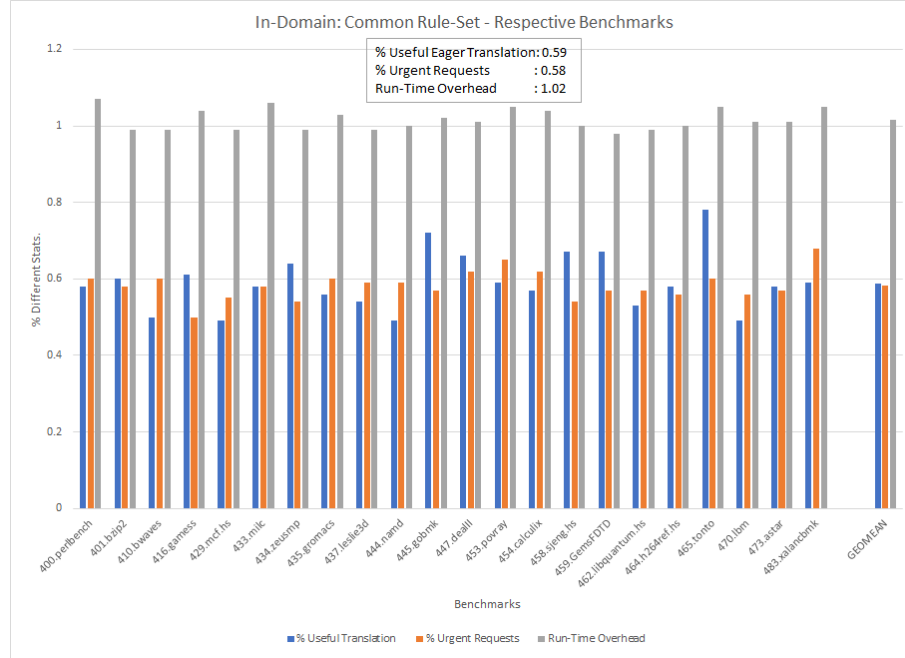


Figure 8.3. Comparison of run-time stats for the test input runs, applying *common* rule-sets with *eager translation*.

Same is the case with the data for *common* rule-set, shown in figure 8.3 with an overhead for most of the benchmarks. On taking a closer look, we can see the distribution of dominant patterns from *certain* and *possible* rule-set data are more balanced in the *common* rule-set, as *% useful eager translation* and *% urgent requests* are very close. That is probably due to the fact that the common rule-set is derived from both certain and possible rule-sets, by simply taking the rules common in both those rule-sets, generated by the LEM1 algorithm. For instance, in certain rule-set, there are **6** outliers where useful translation was less than urgent request percentage, out of the complete set of **22** benchmarks. But, in common rule-set those outliers of certain rule-set (also the dominant pattern of possible rule-set) scaled to **11 out of 22**, leaving the

other half with the dominant pattern of certain rule-set.

Certain Rule-Set		
Stats.	Old Attributes	New Attributes
% Useful Translation	0.50	0.61
% Urgent Requests	0.33	0.56
Run-Time Overhead	0.66	0.91
Possible Rule-Set		
Stats.	Old Attributes	New Attributes
% Useful Translation	0.63	0.57
% Urgent Requests	0.58	0.59
Run-Time Overhead	0.82	1.03
Common Rule-Set		
Stats.	Old Attributes	New Attributes
% Useful Translation	0.63	0.59
% Urgent Requests	0.58	0.58
Run-Time Overhead	0.83	1.02

Table 8.1. *Old Attr. vs New Attr.* consolidated run-time stats for all *SPEC 2006* benchmark in-domain runs with their respective rule-sets

Next will discuss consolidated stats of data gathered with an earlier set of data attributes, before the current set (sample data records in table 7.2 of chapter 7). The earlier set did not have the attributes: *dcbJumpRange*, *dcbJumpDir*, which log the conditional branch jump range and direction respectively. The table 8.1 has the comparison of consolidated stats between old and new attributes for all three stats and all three rule-sets.

At first glance, it might seem like the old attributes have better stats, as clearly the *run-time overhead* and *% useful translation* are better for *possible* and *common* rule-sets with nearly the same *% urgent requests*. For the *certain*

rule-set, the *run-time overhead* is much better for old attributes (*34% speed-up*) than the current set (*9% speed-up*), even though the *% useful translation* is 11% less, which is our main reason to introduce heuristics. The ideal case discussed in section 5.2 of chapter 5, where all the required translations are known ahead of time and translated before application execution, there was significantly high run-time speed-up at *95.5%* and urgent requests at a low of *0.1%*. So, even with the best speed-up of *34%* and low percentage of urgent requests at *33%* for certain rule-set with old-attributes, there is still potential for improvement. However, those stats are for the ideal case and is also in the simulated thread setting where there is absolutely no thread synchronization.

To reiterate on the sub-set of data records used to generate certain and possible rule-sets for inconsistent data-sets gathered in training runs (section 7.1 in chapter 7), the input data for certain rule-sets is a sub-set of consistent data records, within the inconsistent set. So, the rules generated have higher certainty in terms of the branch prediction, compared to possible rules generated from a super-set of records with possibly multiple decision values for the same combination of data attribute values. Possible rules have lesser certainty with respect to branch prediction. Depending on the concept (class of decision value), conflict is resolved by assigning the respective decision value to the deviant data record. Because the input data for rule-induction in that case is tailored to achieve consistency (and is different than the original data), the rules may not always make accurate predictions and may not mark up to the quality of certain rules. The idea behind generating approximated certain and possible rules is to unlock hidden rules that otherwise can't be seen in the original inconsistent

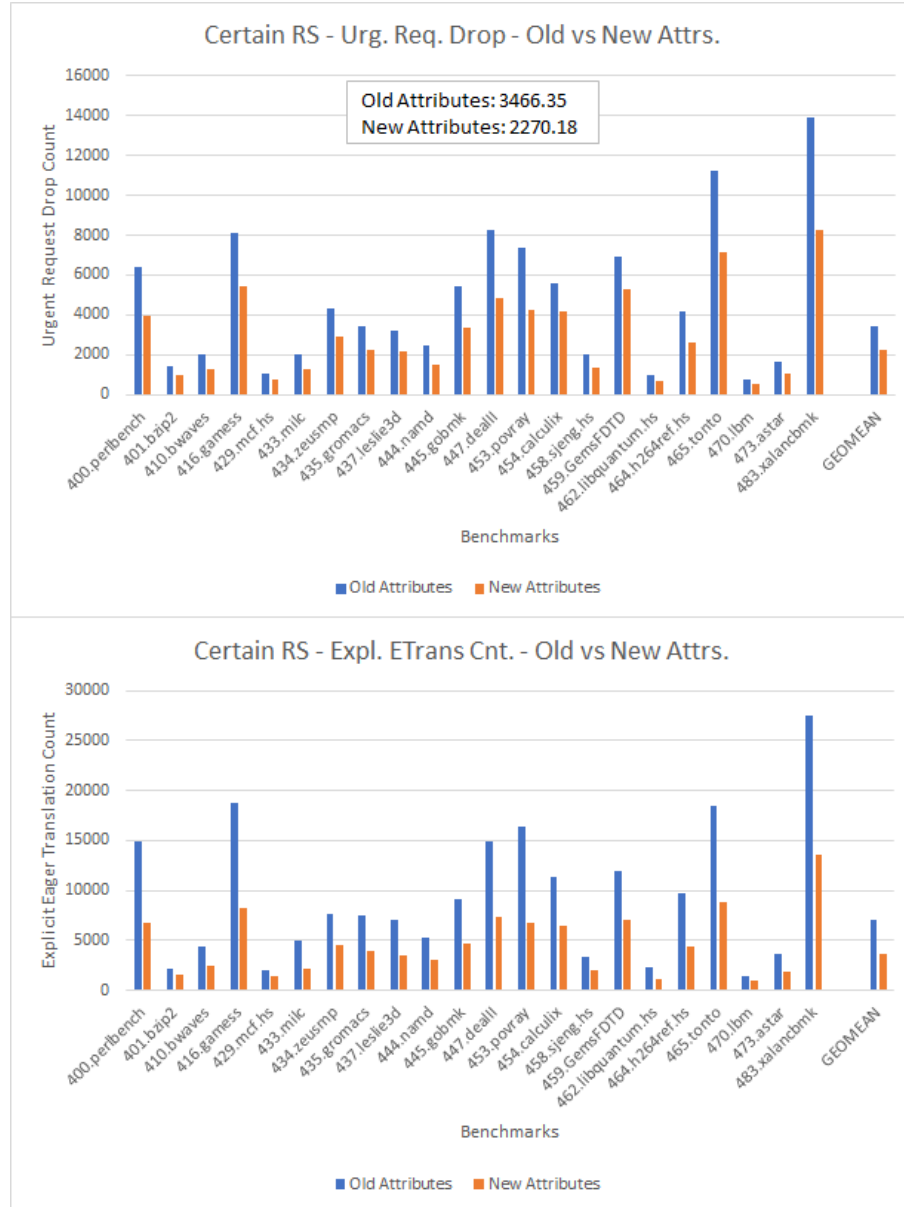


Figure 8.4. *Urgent Request Drop Count, Explicit Eager Translation Count* stats when applying *certain* rule-sets with *eager* translation.

data-set as a whole.

Now, coming back to the comparison of stats between old and new attributes, even though the possible, common rule-set data is better for old attributes, for the certain rule-set there is an improvement in the *% useful translation*. It can also be inferred that with more detail in data through additional data attributes, the eager translation can be made more selective which in turn can promote its usefulness with an obvious trade-off of increased urgent requests, atleast for rule-sets in this case from LEM1. As seen earlier, the certain rule-set runs did manage to get some speed-up with increased urgent requests induced by the increased selectiveness in eager translation. But, the speed-up will hit a wall at some point and may not be linear with more data attributes in the input data-sets for rule-induction. From that point on, the application would be limited by the degree of acceptable overhead. Based on discussion of earlier stats, the prediction accuracy of certain rules generated by LEM1 algorithm is not quite high. But given the process involved in the making of certain and other type of rule-sets, the generic quality of certain rules would be better, eventhough the overall quality might be less depending on the rule-induction algorithm.

This comparison between the two sets of data attributes also gives an important insight to the decision on when to go for extra detail in data and when to settle for a current data attribute set. As the data pattern suggests that more attributes can be tied to added selectiveness, which in turn can reduce the size of translated code, it depends on memory affordability of the application platform. Based on the certain rule-set stats, with lesser attributes, the eager

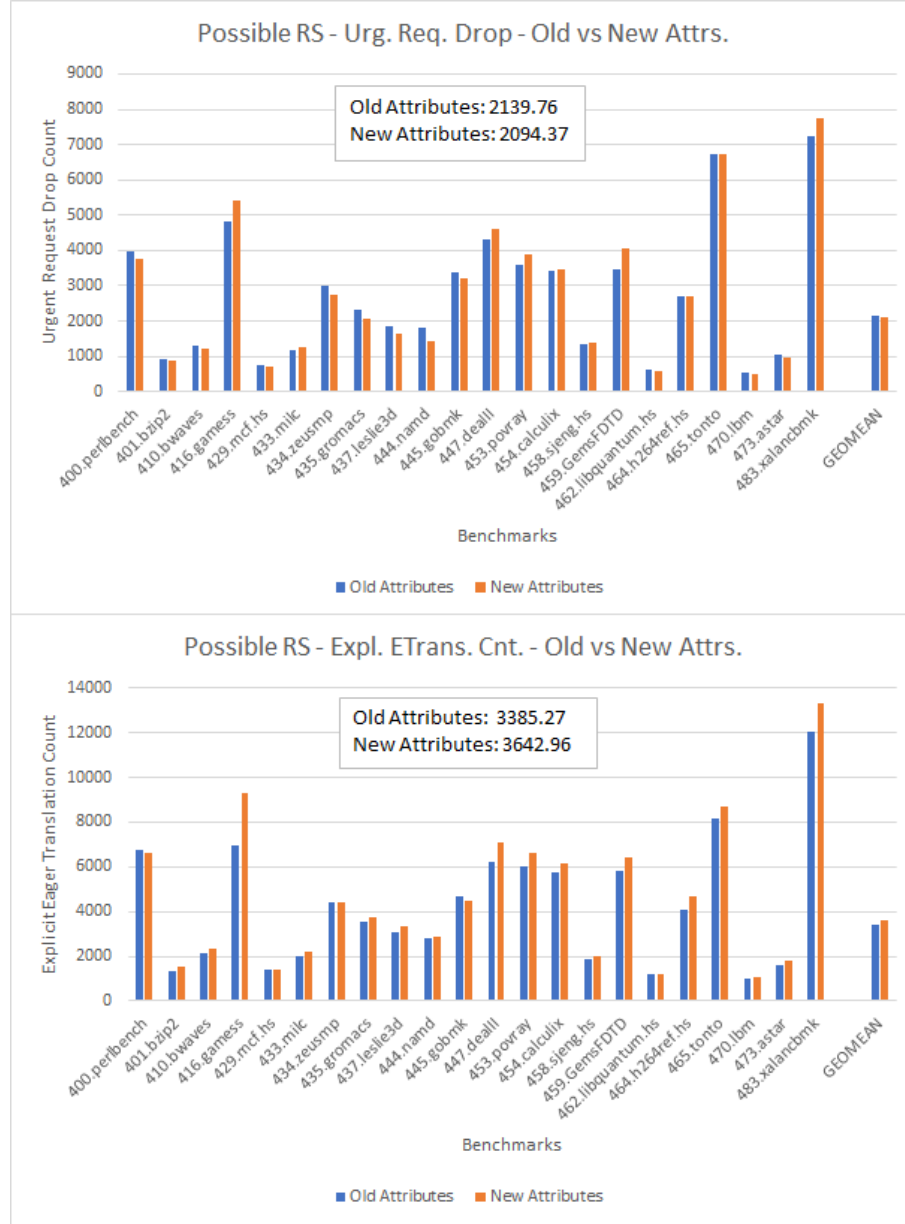


Figure 8.5. *Urgent Request Drop Count, Explicit Eager Translation Count* stats when applying *possible* rule-sets with *eager translation*.

translation is relatively more (less selective) leading to relatively lower urgent requests (smoother application run) and thereby good application speed-up. But, the extra eager translation means more translated code and more requirement for memory in code cache. On the other hand, with 2 additional attributes, the eager translation has become selective, meaning relatively lesser translated code ahead of time that bumped-up the urgent requests (stalls) and thereby a relatively lower speed-up. In this case, the memory requirements for translated code are relatively less, but, also the gains in speed are less even when there is an overall speed-up.

It indicates that when the priority is memory, more attributes can improve the precision and make the eager translation in compiler thread more selective (less eager translations) at the expense of speed-up. But, if speed-up is of priority, less attributes would be the way with reduced precision (reduced translation selectiveness) and relatively more eager translation to ensure much smoother run with reduced urgent requests. So, as mentioned above, choice of data attributes and thereby the rule-sets depends on the needs of application to which the rule-set/heuristics are applied at run-time. Better to have multiple rule-sets generated from data-sets with different attribute sets, to be used with a given application context.

More precise stats in figures 8.4, 8.5, 8.6 for *certain, possible and common* rule-sets respectively, will help in understanding the effect of data attributes on *% useful translation* which is the ratio: *urgent request drop count/explicit eager translation count*. That urgent request drop count is the drop in urgent requests

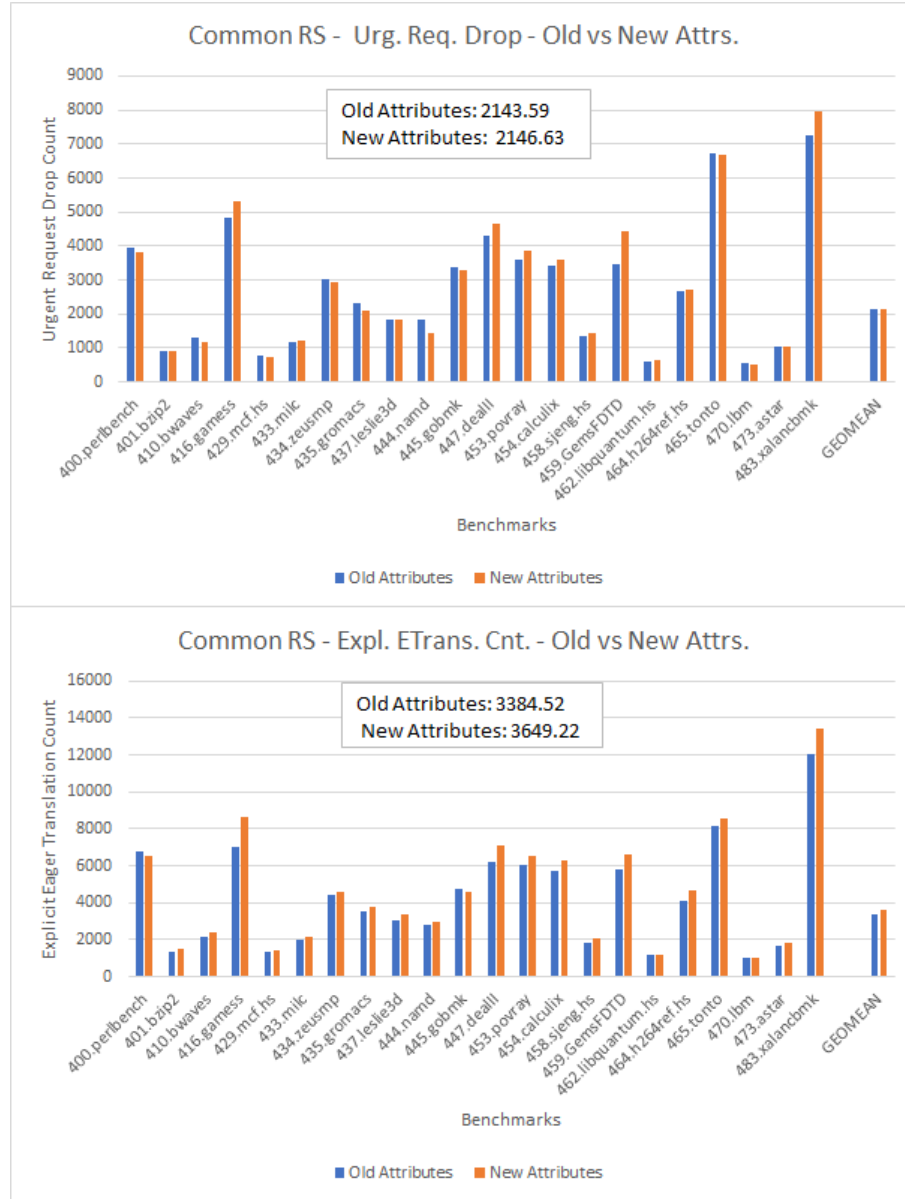


Figure 8.6. *Urgent Request Drop Count, Explicit Eager Translation Count* stats when applying *common* rule-sets with *eager* translation.

from application thread, compared to the baseline requests in a normal application run under DynamoRIO without any eager translation or heuristics.

In figure 8.4 for certain rule-set, both the *urgent request drop count* and *explicit eager translation count* for new attribute set are less compared to the old attribute set. It is a clear indication that the eager translation is more selective with the average (geomean) explicit eager translation count at *3697.11* for new attributes, which is much less than the old attribute count at *6978.07*. That also led to an increased urgent requests, as the average (geomean) drop in urgent requests (compared to the normal baseline count) is also less for new attributes at *2270.18*, compared to *3466.35* with old attributes. That trend shows the lack in expected prediction accuracy (leading to more requests or reduced drop in requests) and is also uniform in all the benchmarks. The closeness between the urgent request drop counts of new and old attributes, accompanied by the big drop in explicit eager translation count caused the improved *% useful eager translation*.

With the *possible and common* rule-sets, as shown in figures 8.5, 8.6 respectively, on an average (geomean) the urgent request drop is very close for both old and new attributes, but, the explicit eager translation count is relatively higher causing a decrease in the *% useful translation*. So, the heuristics generated in those two cases from data-sets with extra attributes/precision couldn't make the eager translation more selective, illustrating a lower quality of rules in comparison with certain rule-set.

Again, this whole dynamic between selectiveness in eager translation and spike in urgent requests is due to the low quality rules leading to low branch prediction accuracy. It would be a different story with higher prediction accuracy, as there can be a reduction in urgent requests in addition to reduced translated code due to the selectiveness induced by heuristics. Data mining/machine learning techniques are more of an art at this point than an exact science. Things that might work for one system may not work that well for others. Hence, there is more work to be done toward formulating quality data-sets and exploring better rule-induction systems that can induce quality rules/heuristics with better accuracy.

(ii) Multi-input benchmark rule-sets:

This configuration is only for benchmarks with multiple inputs. Only 3 benchmarks are qualified: *400.perlbench*, *401.bzip2*, *445.gobmk*. Rule-set generated for one benchmark input is applied to other input runs in the same benchmark. Data-set with old-attributes has been used for the rule-induction. The table 8.2 shows the stats for those three benchmarks.

On an average (geomean) the *certain rule-set* has better results in terms of *run-time speed-up* at 36% and a low % *urgent requests* at 38%, eventhough the % *useful translation* is at 57%. The *possible and common* rule-sets have a better % *useful translation* at 64% and 65% respectively. It suggests that the eager translation was more selective in *possible, common* rule-sets leading to better % *useful translation* compared to *certain* rule-set, with the rise in % *urgent requests* as a trade-off. This is inline with the results found in prior

Certain Rule-Set			
Benchmark	% Useful Translation	% Urgent Requests	Run-Time Overhead
400.perlbench	0.42	0.36	0.33
401.bzip2	0.71	0.48	0.92
445.gobmk	0.61	0.33	0.85
Geomean	0.57	0.38	0.64
Possible Rule-Set			
Benchmark	% Useful Translation	% Urgent Requests	Run-Time Overhead
400.perlbench	0.57	0.59	0.61
401.bzip2	0.67	0.59	0.93
445.gobmk	0.70	0.56	0.90
Geomean	0.64	0.58	0.80
Common Rule-Set			
Benchmark	% Useful Translation	% Urgent Requests	Run-Time Overhead
400.perlbench	0.57	0.59	0.61
401.bzip2	0.67	0.59	0.94
445.gobmk	0.70	0.56	0.91
Geomean	0.65	0.58	0.80

Table 8.2. Multi-input benchmark run configuration: Rule-set of one input applied to another in same benchmark

configuration where benchmarks were run with their respective rule-sets.

Since the certain rules are generated from a more concrete set of data records with well mapped branch prediction values and the results show applications with more data and increased *% useful eager translation*, the new attribute data-sets and corresponding certain rule-sets have been used for rest of the run configurations to apply heuristics.

(iii) Logical domains *SPEC_INT*, *SPEC_FLOAT* - Inner Random Rule-Sets:

As suggested in their names, SPEC 2006 integer and floating-point benchmarks constitute *SPEC_INT* and *SPEC_FLOAT* domains respectively. For this

configuration, rule-set of one benchmark is run against others within its respective logical domain. Those two are "*logical in-domains*", but purpose of this run-configuration is to still explore how well or bad a given benchmark or data-set would do with a different rule-set or benchmark respectively. That is to get an insight in to whether an existing rule-set generated for a certain application, be readily used with a random new application without the need to generate its own rule-set with training data (through some training runs).

Since each of the benchmarks were run with different rule-sets not specifically tailored to their runs, the patterns are unpredictable and involves doing lot of benchmark runs in different possible configurations, collecting data and looking for useful inferences. The data in tables 8.3, 8.4, 8.5 is for *% useful eager translation, % urgent requests, run-time overhead* respectively, in *SPEC_INT* logical in-domain runs with certain rule-sets. To compare the 3 run-time stats with those gathered from runs with respective benchmark rule-sets (discussed in previous run-configuration), all the data points in each of the tables are ratios of stats in this configuration to the ones in respective rule-set configuration (baseline stat).

To know how well a benchmark rule-set works with other domain members and also to gauge how receptive a benchmark is to the randomness in applied rule-sets, two new stats are introduced, namely: *Rule-Set Effectiveness and Benchmark Compatibility* for run-configurations involving rule-sets of different benchmarks. Those two stats don't involve any fancy computations. They are pretty much averages (geomeans in the tables) of stats recorded for different

Benchmark	400.perlbench	401.bzip2	429.mcf.hs	445.gobmk	458.sjeng.hs
400.perlbench	-	0.97	0.97	0.93	0.97
401.bzip2	0.94	-	1.06	0.94	0.95
429.mcf.hs	0.94	0.98	-	0.92	0.96
445.gobmk	1.00	0.99	1.00	-	1.00
458.sjeng.hs	0.99	1.00	0.99	0.99	-
462.libquantum.hs	0.82	0.78	0.85	0.77	0.75
464.h264ref.hs	1.00	0.97	0.98	0.97	1.00
473.astar	1.04	1.02	1.02	1.02	0.98
483.xalancbmk	0.95	0.98	0.97	0.97	0.95
GEOMEAN(Rule-Set Effectiveness)	0.96	0.96	0.98	0.93	0.94

Benchmark	462.libquantum.hs	464.h264ref.hs	473.astar	483.xalancbmk	GEOMEAN(Benchmark Compatibility)
400.perlbench	1.02	0.98	0.95	1.02	0.97
401.bzip2	1.08	0.94	1.03	1.03	0.99
429.mcf.hs	1.10	0.96	1.04	1.00	0.99
445.gobmk	1.03	0.99	0.99	1.01	1.00
458.sjeng.hs	1.06	0.99	1.04	1.00	1.01
462.libquantum.hs	-	0.73	0.92	0.83	0.80
464.h264ref.hs	1.07	-	1.00	0.98	1.00
473.astar	1.12	1.02	-	1.07	1.03
483.xalancbmk	1.02	0.97	0.97	-	0.97
GEOMEAN(Rule-Set Effectiveness)	1.06	0.94	0.99	0.99	

Table 8.3. *% Useful Etrans. stat ratios* (with respective rule-set stats) for *SPEC 2006* benchmark runs: *SPEC-INT* logical in-domain with certain rule-set

benchmark rule-sets and runs, which can be seen in the last row and column outside the benchmark data matrix within the tables 8.3, 8.4, 8.5. As this configuration is to capture stats with benchmarks run against rule-sets of other benchmarks within respective logical domains and afore mentioned tables are for *SPEC-INT*, the first column and row of listed benchmarks belong to just that domain. The diagonal entries are blank as those would be the runs with respective rule-sets, that are not part of this run-configuration.

The table 8.3 has ratios for *% useful eager translation*. Any value of *ratio* > 1 is considered an improvement compared to the stat with respective rule-set. As indicated in the table, the rule-set of benchmark *462.libquantum.hs* has an average (geomean) value of *1.06* in the last row with a bump of *6%*, proving to be effective across random benchmarks within the *SPEC-INT* domain. There are other benchmarks that are close as well, like *473.astar*, *483.xalancbmk* at

0.99. Similarly, the *benchmark compatibility* can be seen in the last column and oddly the lowest value of 0.80 (20% drop) is for 462.libquantum.hs whose rule-set seemed to worked well with other benchmarks. The benchmark 473.astar has the best result with 3% improvement when run with random rule-sets, but, other benchmarks are quite close too. So, relatively most benchmarks seem to work well with random rule-sets.

Benchmark	400.perlbench	401.bzip2	429.mcf.hs	445.gobmk	458.sjeng.hs
400.perlbench	-	1.03	1.00	1.02	1.02
401.bzip2	1.09	-	0.96	1.05	1.07
429.mcf.hs	1.08	1.04	-	1.04	1.08
445.gobmk	1.04	1.05	1.02	-	1.02
458.sjeng.hs	1.00	1.00	0.98	1.00	-
462.libquantum.hs	1.11	1.11	1.06	1.11	1.15
464.h264ref.hs	0.98	0.98	0.95	0.98	0.96
473.astar	1.00	0.97	0.97	0.97	1.00
483.xalancbmk	1.04	1.01	1.01	1.03	1.03
GEOMEAN(Rule-Set Effectiveness)	1.04	1.02	0.99	1.02	1.04

Benchmark	462.libquantum.hs	464.h264ref.hs	473.astar	483.xalancbmk	GEOMEAN(Benchmark Compatibility)
400.perlbench	0.98	1.03	1.05	0.93	1.01
401.bzip2	0.96	1.11	1.00	0.95	1.02
429.mcf.hs	0.94	1.08	1.02	0.98	1.03
445.gobmk	0.96	1.05	1.05	0.98	1.02
458.sjeng.hs	0.91	1.04	0.98	0.96	0.98
462.libquantum.hs	-	1.15	1.07	1.06	1.10
464.h264ref.hs	0.89	-	0.96	0.95	0.96
473.astar	0.91	1.00	-	0.91	0.96
483.xalancbmk	1.01	1.03	1.03	-	1.03
GEOMEAN(Rule-Set Effectiveness)	0.95	1.06	1.02	0.96	

Table 8.4. % Urgent Request stat ratios (with respective rule-set stats) for *SPEC 2006* benchmark runs: *SPEC_INT* logical in-domain with certain rule-set

Table 8.4 has the ratios for % urgent requests. In this case, it is desired to have a value below 1, an indication of reduced urgent requests or smoother application run. The best rule-set again is of the benchmark 462.libquantum.hs with *rule-set effectiveness* value at 0.95 (5% drop). Most compatible benchmarks with different rule-sets are: 464.h264ref.hs, 473.astar with a value of 0.96 (4% drop). Worst is again 462.libquantum.hs with 10% increase (last column). Rest of the benchmarks are close.

Benchmark	400.perlbench	401.bzip2	429.mcf.hs	445.gobmk	458.sjeng.hs
400.perlbench	-	0.76	0.74	0.85	0.80
401.bzip2	1.11	-	1.02	1.05	1.01
429.mcf.hs	1.15	1.07	-	1.10	1.00
445.gobmk	1.01	0.95	0.95	-	0.96
458.sjeng.hs	1.02	0.99	0.99	1.01	-
462.libquantum.hs	2.00	1.14	1.07	1.49	1.30
464.h264ref.hs	1.01	1.00	0.99	1.01	1.01
473.astar	1.05	1.01	1.01	1.03	1.02
483.xalancbmk	1.01	0.77	0.77	0.85	0.82
GEOMEAN(Rule-Set Effectiveness)	1.14	0.95	0.93	1.04	0.98

Benchmark	462.libquantum.hs	464.h264ref.hs	473.astar	483.xalancbmk	GEOMEAN(Benchmark Compatibility)
400.perlbench	0.70	0.92	0.79	1.03	0.82
401.bzip2	0.98	1.05	1.02	1.13	1.05
429.mcf.hs	0.99	1.04	1.07	1.15	1.07
445.gobmk	0.93	0.98	0.96	1.02	0.97
458.sjeng.hs	0.96	0.99	0.99	1.03	1.00
462.libquantum.hs	-	1.65	1.15	2.13	1.45
464.h264ref.hs	1.00	-	1.00	1.02	1.01
473.astar	1.01	1.04	-	1.05	1.03
483.xalancbmk	0.75	0.89	0.79	-	0.83
GEOMEAN(Rule-Set Effectiveness)	0.91	1.05	0.97	1.16	

Table 8.5. *Run-Time Overhead stat ratios* (with respective rule-set stats) for *SPEC 2006* benchmark runs: *SPEC_INT* logical in-domain with certain rule-set

Last stat is *run-time overhead* with ratios in the table 8.5. A ratio below 1 is also desirable for this stat, compared to the respective rule-set stat, as it would mean a guaranteed better run-time performance with random heuristics. The best rule-set for the third time is *462.libquantum.hs* with 9% speed-up, and the worst overhead for *400.perlbench*, *483.xalancbmk* at 14% and 16% respectively. In terms of *benchmark compatibility*, the trend is flipped with best compatibility for *400.perlbench*, *483.xalancbmk* with speed-ups at 18%, 17% respectively and the worst compatibility for *462.libquantum.hs* with an overhead of 45%. The stats of other benchmarks are quite close.

Next is the consolidated stats for each benchmark to know which rule-sets are more effective and which benchmarks are more compatible, as a whole, over a spectrum of benchmarks. Data-points are marked as "X" for fillers from

Benchmark	400.perlbench	401.bzip2	429.mcf.hs	445.gobmk	458.sjeng.hs	462.libquantum.hs
400.perlbench	-	X	X	X	X	X
401.bzip2	X	-	X	X	X	X
429.mcf.hs	X	X	-	X	X	X
445.gobmk	X	X	X	-	X	X
458.sjeng.hs	X	X	X	X	-	X
462.libquantum.hs	X	X	X	X	X	-
464.h264ref.hs	X	X	X	X	X	X
473.astar	X	X	X	X	X	X
483.xalancbmk	X	X	X	X	X	X
<i>GEOMEAN(Rule-Set Effectiveness)</i>						
%Useful Etrans.	0.96	0.96	0.98	0.93	0.94	1.06
%Urgent Requests	1.04	1.02	0.99	1.02	1.04	0.95
Run-Time Overhead	1.14	0.95	0.93	1.04	0.98	0.91
				<i>GEOMEAN(Benchmark Compatibility)</i>		
Benchmark	464.h264ref.hs	473.astar	483.xalancbmk	%Useful Etrans.	%Urgent Requests	Run-Time Overhead
400.perlbench	X	X	X	0.97	1.01	0.82
401.bzip2	X	X	X	0.99	1.02	1.05
429.mcf.hs	X	X	X	0.99	1.03	1.07
445.gobmk	X	X	X	1.00	1.02	0.97
458.sjeng.hs	X	X	X	1.01	0.98	1.00
462.libquantum.hs	X	X	X	0.80	1.10	1.45
464.h264ref.hs	-	X	X	1.00	0.96	1.01
473.astar	X	-	X	1.03	0.96	1.03
483.xalancbmk	X	X	-	0.97	1.03	0.83
<i>GEOMEAN(Rule-Set Effectiveness)</i>						
%Useful Etrans.	0.94	0.99	0.99			
%Urgent Requests	1.06	1.02	0.96			
Run-Time Overhead	1.05	0.97	1.16			

Table 8.6. *Consolidated stat ratios* (with respective rule-set stats) for *SPEC 2006* benchmark runs: *SPEC_INT* logical in-domain with certain rule-set

individual stat tables. The rule-set of benchmark *462.libquantum.hs* has the best result for all three stats and benchmark with best overall compatibility is *445.gobmk* with a speed-up of 3% and an even % useful translation, even though there is a 2% increase in the % urgent requests. The information on effective rule-sets would be useful in readily applying heuristics to a new application, as they are known to be effective with random applications. The benchmark compatibility on the other hand can be explored further with respect to the design of the application, code layout and so forth to possibly know why they are more compatible (or not) which can also help us in establishing some coding standards, or the format of compiled code to enable a more standardized use of heuristics and even eager translation in DBT systems with pre-optimized

application code to work with.

Similar analysis can be done for *SPEC_FLOAT* in-domain configuration. The corresponding consolidated stats are in the table 8.10. Best rule-sets are for benchmarks *437.leslie3d*, *459.GemsFDTD*, *470.lbm* with an improved % *useful translation* and *run-time overhead*. Best compatible benchmarks are *465.tonto*, *435.gromacs* with even or improved stats. Individual data can be seen in separate tables 8.7, 8.8, 8.9 for % *useful translation*, % *urgent requests* and *run-time overhead* respectively.

Benchmark	410.bwaves	416.gamess	433.milc	434.zeusmp	435.gromacs	437.leslie3d	444.namd
410.bwaves	-	1.04	1.04	1.04	1.02	1.12	0.92
416.gamess	0.89	-	0.82	0.88	0.86	0.88	0.80
433.milc	1.02	1.00	-	0.98	0.97	1.07	0.93
434.zeusmp	0.94	1.05	0.94	-	0.97	0.97	0.95
435.gromacs	1.02	1.00	0.98	1.00	-	1.04	0.95
437.leslie3d	0.92	0.90	0.93	0.97	0.95	-	0.84
444.namd	1.22	1.14	1.02	1.16	1.00	1.24	-
447.dealII	1.00	0.98	1.03	1.03	1.03	1.02	1.00
453.povray	1.00	0.95	0.98	0.98	0.95	0.97	0.89
454.calculix	0.91	0.91	0.98	0.98	0.95	0.98	0.88
459.GemsFDTD	0.84	0.88	0.92	0.91	0.85	0.96	0.80
465.tonto	0.99	1.00	0.99	1.04	0.99	1.01	0.94
470.lbm	0.89	0.89	0.76	0.89	0.81	1.00	0.78
GEOMEAN(Rule-Set Effectiveness)	0.96	0.98	0.95	0.99	0.94	1.02	0.89

Benchmark	447.dealII	453.povray	454.calculix	459.GemsFDTD	465.tonto	470.lbm	GEOMEAN(Benchmark Compatibility)
410.bwaves	0.96	0.96	1.04	1.15	1.06	1.13	1.04
416.gamess	0.83	0.95	0.92	0.95	0.88	0.88	0.88
433.milc	0.93	0.93	1.00	1.10	1.00	1.09	1.00
434.zeusmp	0.94	0.97	1.03	1.08	1.00	1.03	0.99
435.gromacs	0.96	0.95	1.00	1.05	1.00	1.05	1.00
437.leslie3d	0.82	0.82	0.93	1.05	0.93	0.98	0.92
444.namd	1.04	1.02	1.08	1.22	1.10	1.12	1.11
447.dealII	-	1.00	1.05	1.03	1.03	1.03	1.02
453.povray	0.90	-	1.02	1.02	1.02	1.00	0.97
454.calculix	0.86	0.86	-	1.00	0.92	0.97	0.93
459.GemsFDTD	0.76	0.83	0.92	-	0.80	1.00	0.87
465.tonto	0.95	0.99	1.02	1.04	-	1.01	1.00
470.lbm	0.85	0.80	0.85	0.98	0.89	-	0.86
GEOMEAN(Rule-Set Effectiveness)	0.90	0.92	0.99	1.05	0.97	1.02	

Table 8.7. % *Useful Etrans. stat ratios* (with respective rule-set stats) for *SPEC 2006* benchmark runs: *SPEC_FLOAT* logical in-domain with certain rule-set

8.2.2 Out-Domain

(i) Test input rule-set applied to reference-input runs:

Benchmark	410.bwaves	416.gamess	433.milc	434.zeusmp	435.gromacs	437.leslie3d	444.namd
410.bwaves	-	1.02	0.98	0.98	1.04	0.95	1.09
416.gamess	1.00	-	1.04	1.00	1.08	1.00	1.10
433.milc	0.98	1.02	-	1.00	1.05	0.95	1.07
434.zeusmp	1.06	1.00	1.07	-	1.04	1.02	1.06
435.gromacs	0.98	1.00	0.98	0.98	-	0.93	1.05
437.leslie3d	1.08	1.16	1.08	1.06	1.10	-	1.20
444.namd	0.86	0.95	0.95	0.89	1.00	0.84	-
447.dealII	1.00	1.00	0.97	0.97	0.98	0.98	1.00
453.povray	0.98	1.02	0.98	0.98	1.03	0.98	1.10
454.calculix	1.07	1.07	0.98	0.98	1.04	1.00	1.11
459.GemsFDTD	1.20	1.18	1.14	1.10	1.20	1.04	1.31
465.tonto	0.98	0.98	0.96	0.95	1.00	0.95	1.04
470.lbm	1.10	1.16	1.18	1.12	1.14	1.00	1.18
GEOMEAN(Rule-Set Effectiveness)	1.02	1.04	1.02	1.00	1.06	0.97	1.10

Benchmark	447.dealII	453.povray	454.calculix	459.GemsFDTD	465.tonto	470.lbm	GEOMEAN(Benchmark Compatibility)
410.bwaves	1.09	1.07	1.00	0.91	1.02	0.93	1.00
416.gamess	1.10	1.02	1.00	0.94	1.02	0.96	1.02
433.milc	1.07	1.05	1.02	0.91	1.02	0.93	1.00
434.zeusmp	1.13	1.04	0.98	0.93	1.02	0.96	1.02
435.gromacs	1.03	1.03	0.98	0.91	1.00	0.91	0.98
437.leslie3d	1.24	1.25	1.10	0.96	1.14	1.00	1.11
444.namd	1.00	1.02	0.93	0.82	0.96	0.89	0.92
447.dealII	-	1.00	0.95	0.95	1.00	0.93	0.98
453.povray	1.07	-	0.98	0.95	0.98	0.97	1.00
454.calculix	1.11	1.12	-	0.96	1.09	0.96	1.04
459.GemsFDTD	1.35	1.24	1.14	-	1.31	1.00	1.18
465.tonto	1.02	0.98	0.96	0.93	-	0.93	0.97
470.lbm	1.16	1.22	1.14	1.00	1.14	-	1.13
GEOMEAN(Rule-Set Effectiveness)	1.11	1.08	1.01	0.93	1.05	0.95	

Table 8.8. % Urgent Request stat ratios (with respective rule-set stats) for *SPEC 2006* benchmark runs: *SPEC_FLOAT* logical in-domain with certain rule-set

Benchmark	410.bwaves	416.gamess	433.milc	434.zeusmp	435.gromacs	437.leslie3d	444.namd
410.bwaves	-	1.02	1.01	1.01	1.02	1.01	1.01
416.gamess	0.71	-	0.75	0.79	0.89	0.75	0.78
433.milc	1.16	1.04	-	1.00	1.01	1.00	1.01
434.zeusmp	1.03	1.01	1.00	-	0.99	0.99	1.00
435.gromacs	0.95	1.08	0.95	0.98	-	0.94	0.98
437.leslie3d	1.00	1.03	1.00	1.00	1.01	-	1.01
444.namd	0.99	1.01	0.99	1.00	1.00	0.99	-
447.dealII	0.97	0.99	0.97	0.97	0.99	0.97	0.97
453.povray	0.88	1.02	0.88	0.90	0.94	0.88	0.93
454.calculix	0.91	1.14	0.86	0.90	0.98	0.88	0.96
459.GemsFDTD	1.07	1.10	1.01	1.01	1.05	0.99	1.07
465.tonto	0.77	0.97	0.77	0.79	0.85	0.77	0.80
470.lbm	1.01	1.03	1.00	1.01	1.01	1.00	1.00
GEOMEAN(Rule-Set Effectiveness)	0.95	1.04	0.93	0.94	0.98	0.93	0.96

Benchmark	447.dealII	453.povray	454.calculix	459.GemsFDTD	465.tonto	470.lbm	GEOMEAN(Benchmark Compatibility)
410.bwaves	1.02	1.02	1.02	1.01	1.02	1.00	1.01
416.gamess	1.12	1.04	0.96	0.77	1.16	0.59	0.84
433.milc	1.05	1.04	1.05	1.01	1.05	0.98	1.03
434.zeusmp	1.01	1.00	1.00	0.99	1.01	0.98	1.00
435.gromacs	1.09	1.09	1.05	0.95	1.11	0.89	1.00
437.leslie3d	1.04	1.03	1.02	1.01	1.04	0.99	1.02
444.namd	1.01	1.01	1.01	0.99	1.01	0.98	1.00
447.dealII	-	0.99	0.98	0.97	1.00	0.96	0.98
453.povray	1.04	-	0.96	0.89	1.02	0.82	0.93
454.calculix	1.17	1.16	-	0.88	1.17	0.78	0.97
459.GemsFDTD	1.11	1.14	1.04	-	1.10	0.91	1.05
465.tonto	0.98	0.96	0.90	0.79	-	0.71	0.83
470.lbm	1.02	1.04	1.01	1.00	1.02	-	1.01
GEOMEAN(Rule-Set Effectiveness)	1.05	1.04	1.00	0.93	1.06	0.87	

Table 8.9. Run-Time Overhead stat ratios (with respective rule-set stats) for *SPEC 2006* benchmark runs: *SPEC_FLOAT* logical in-domain with certain rule-set

Benchmark	410.bwaves	416.gamess	433.milc	434.zeusmp	435.gromacs	437.leslie3d	444.namd	447.dealII
410.bwaves	-	X	X	X	X	X	X	X
416.gamess	X	-	X	X	X	X	X	X
433.milc	X	X	-	X	X	X	X	X
434.zeusmp	X	X	X	-	X	X	X	X
435.gromacs	X	X	X	X	-	X	X	X
437.leslie3d	X	X	X	X	X	-	X	X
444.namd	X	X	X	X	X	X	-	X
447.dealII	X	X	X	X	X	X	X	-
453.povray	X	X	X	X	X	X	X	X
454.calculix	X	X	X	X	X	X	X	X
459.GemsFDTD	X	X	X	X	X	X	X	X
465.tonto	X	X	X	X	X	X	X	X
470.lbm	X	X	X	X	X	X	X	X
<i>GEOMEAN(Rule-Set Effectiveness)</i>								
%Useful Etrans.	0.96	0.98	0.95	0.99	0.94	1.02	0.89	0.90
%Urgent Requests	1.02	1.04	1.02	1.00	1.06	0.97	1.10	1.11
Run-Time Overhead	0.95	1.04	0.93	0.94	0.98	0.93	0.96	1.05
<i>GEOMEAN(Benchmark Compatibility)</i>								
Benchmark	453.povray	454.calculix	459.GemsFDTD	465.tonto	470.lbm	%Useful Etrans.	%Urgent Requests	Run-Time Overhead
410.bwaves	X	X	X	X	X	1.04	1.00	1.01
416.gamess	X	X	X	X	X	0.88	1.02	0.84
433.milc	X	X	X	X	X	1.00	1.00	1.03
434.zeusmp	X	X	X	X	X	0.99	1.02	1.00
435.gromacs	X	X	X	X	X	1.00	0.98	1.00
437.leslie3d	X	X	X	X	X	0.92	1.11	1.02
444.namd	X	X	X	X	X	1.11	0.92	1.00
447.dealII	X	X	X	X	X	1.02	0.98	0.98
453.povray	-	X	X	X	X	0.97	1.00	0.93
454.calculix	X	-	X	X	X	0.93	1.04	0.97
459.GemsFDTD	X	X	-	X	X	0.87	1.18	1.05
465.tonto	X	X	X	-	X	1.00	0.97	0.83
470.lbm	X	X	X	X	-	0.86	1.13	1.01
<i>GEOMEAN(Rule-Set Effectiveness)</i>								
%Useful Etrans.	0.92	0.99	1.05	0.97	1.02			
%Urgent Requests	1.08	1.01	0.93	1.05	0.95			
Run-Time Overhead	1.04	1.00	0.93	1.06	0.87			

Table 8.10. *Consolidated stat ratios* (with respective rule-set stats)
for *SPEC 2006* benchmark runs: *SPEC_FLOAT* logical in-domain
with certain rule-set

This work involves benchmark runs with test inputs. But, to have a variety in terms of application data, the *certain* rule-sets generated for test inputs have been run with reference input to make another logical out-domain. The *run-time overhead* stat is not included in the data analysis of this run-configuration as reference inputs are long running programs and designed to overcome start-up overhead through code reusability over a longer period of application execution. Its different for test inputs as they are short running programs and start-up overhead sticks. So, comparison of ref and test input run times to showcase the effects of heuristics wouldn't be helpful.

The figure 8.7 has the comparison of test and ref data with *% useful translation*

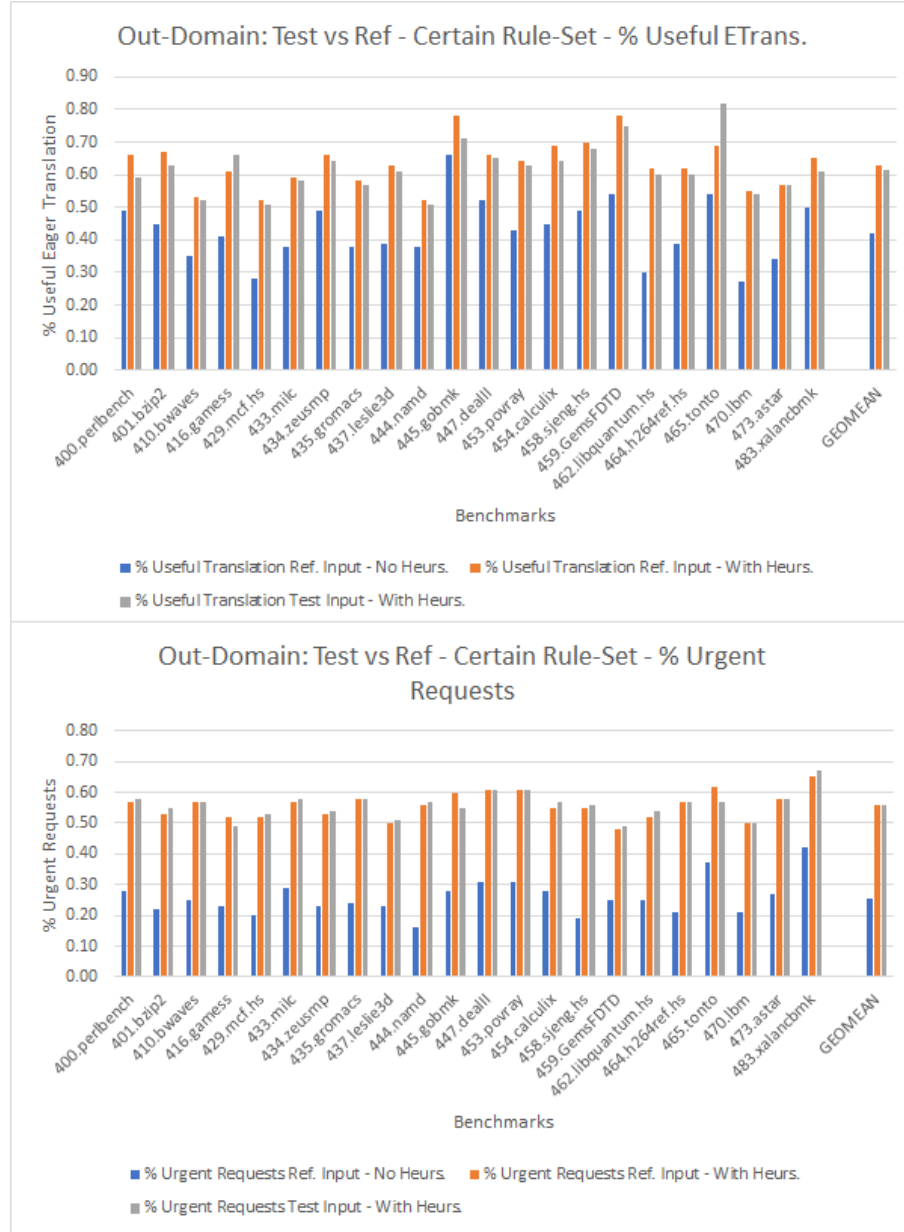


Figure 8.7. Comparison of % *Useful Eager Translation* and % *Urgent Requests* for the test vs ref inputs applying *certain* rule-sets with *eager translation*.

and % *urgent requests*. Additional eager translation data has also been gathered for ref input runs with heuristics plugged-out. That is to know the influence of test-input heuristics on ref-input runs. Clearly, there is improvement in % *useful translation* with heuristics (42% -> 63%), with a big spike in % *urgent requests* (25% -> 56%) as seen in prior configurations, highlighting the low quality of rules/heuristics. That pattern is uniform for all the benchmark with no outliers.

But, the initial overhead with increased urgent requests would be amortized during the longer runs of ref-inputs. So, the reduction in overall eager translation by the selectiveness induced with certain rule-sets would again reduce the footprint of translated code. It can open up more platforms for DBTs where memory is a constraint. Atleast for long running programs like the ref-inputs, the heuristics will definitely help with size of translated target code, irrespective of the quality of generated heuristics. Similar pattern can be seen with the other two rule-sets in figures 8.8, 8.9.

(ii) Logical domains *SPEC_INT*, *SPEC_FLOAT* - Outer Random Rule-Sets:

This configuration involves *certain* rule-sets of one domain being applied for benchmark runs in another domain. Gathered data is quite extensive to give us an insight on how well a rule-set can work with random applications. More precisely, the rule-sets of benchmarks in *SPEC_INT* would be applied for runs of benchmarks in *SPEC_FLOAT* and vice-versa for a more broader view of *rule-set effectiveness and benchmark compatibility*. Those are the same two stats used in the corresponding in-domain configuration for *SPEC_INT*, *SPEC_FLOAT*, that

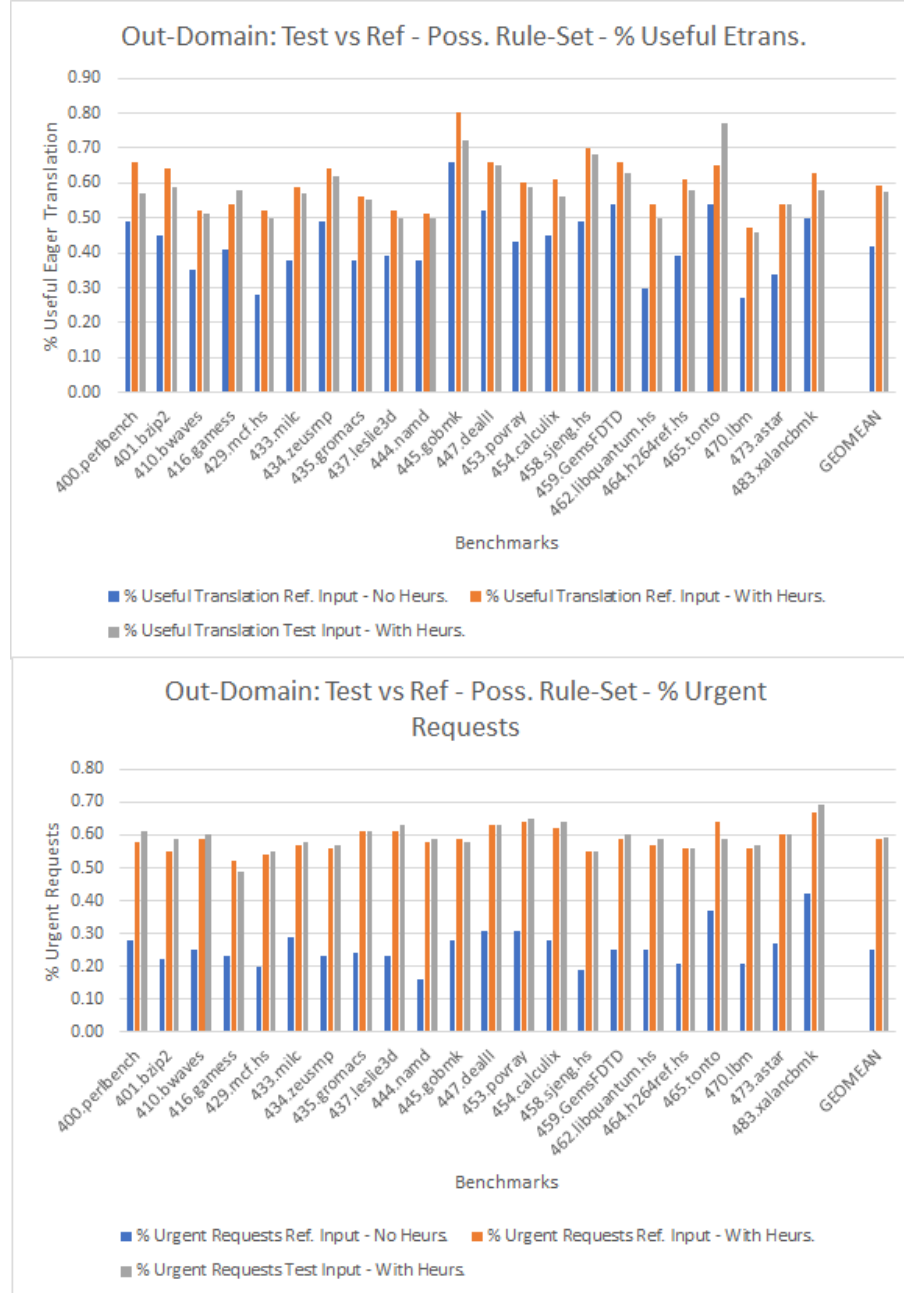


Figure 8.8. Comparison of % *Useful Eager Translation* and % *Urgent Requests* for the test vs ref inputs applying *possible* rule-sets with *eager translation*.

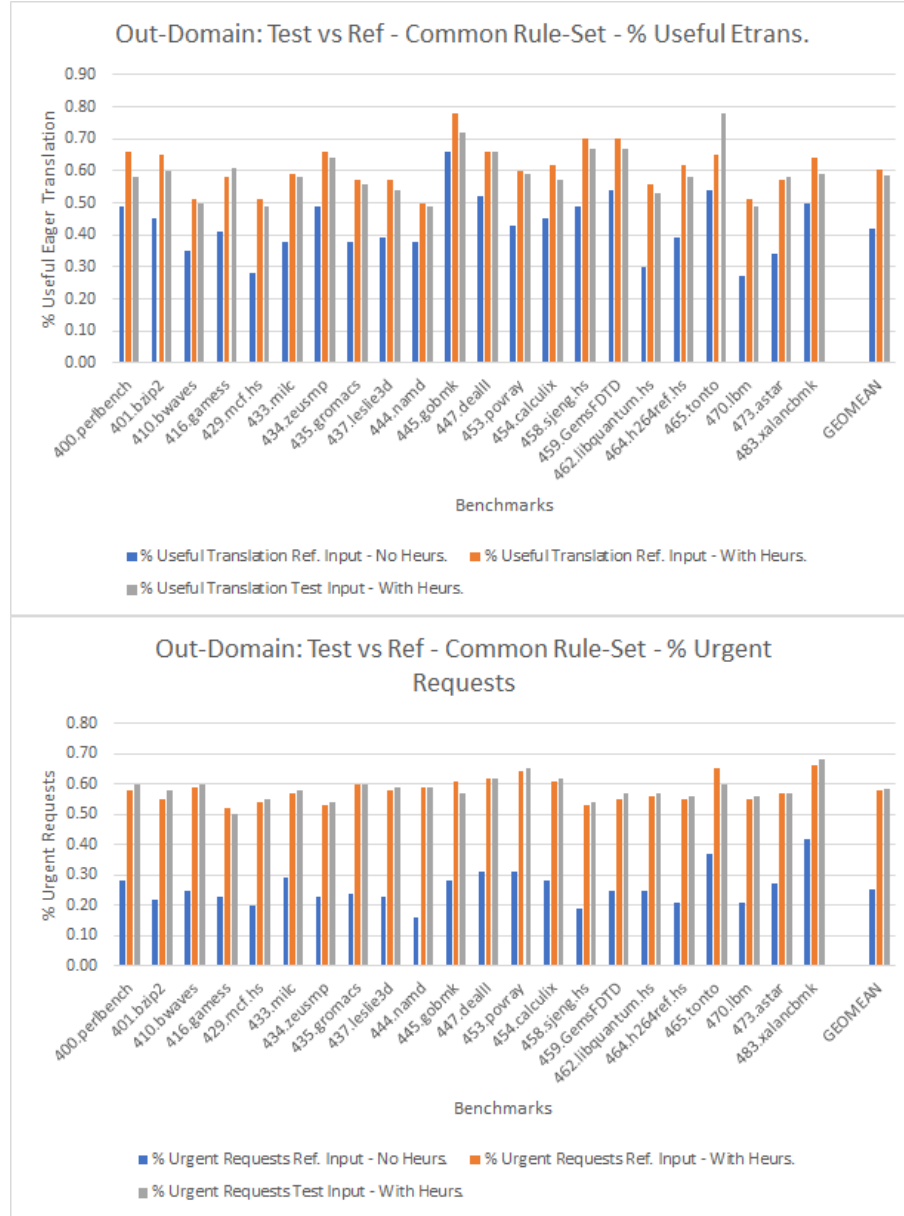


Figure 8.9. Comparison of % *Useful Eager Translation* and % *Urgent Requests* for the test vs ref inputs applying *common* rule-sets with *eager translation*.

could rate rule-sets and benchmarks.

Benchmark	410.bwaves	416.gamess	433.milc	434.zeusmp	435.gromacs	437.leslie3d	444.namd	447.dealII
400.perlbench	X	X	X	X	X	X	X	X
401.bzip2	X	X	X	X	X	X	X	X
429.mcf.hs	X	X	X	X	X	X	X	X
445.gobmk	X	X	X	X	X	X	X	X
458.sjeng.hs	X	X	X	X	X	X	X	X
462.libquantum.hs	X	X	X	X	X	X	X	X
464.h264ref.hs	X	X	X	X	X	X	X	X
473.astar	X	X	X	X	X	X	X	X
483.xalancbmk	X	X	X	X	X	X	X	X
<i>GEOMEAN(Rule-Set Effectiveness)</i>								
%Useful Etrans.	0.98	0.97	0.94	0.99	0.96	1.03	0.93	0.92
%Urgent Requests	1.01	1.04	1.01	0.99	1.02	0.97	1.06	1.07
Run-Time Overhead	0.97	1.11	0.97	0.99	1.02	0.97	0.99	1.12
<i>GEOMEAN(Benchmark Compatibility)</i>								
Benchmark	453.povray	454.calculix	459.GemsFDTD	465.tonto	470.lbm	%Useful Etrans.	%Urgent Requests	Run-Time Overhead
400.perlbench	X	X	X	X	X	0.98	1.00	0.87
401.bzip2	X	X	X	X	X	1.02	1.00	1.04
429.mcf.hs	X	X	X	X	X	1.00	1.03	1.07
445.gobmk	X	X	X	X	X	1.00	1.02	0.98
458.sjeng.hs	X	X	X	X	X	1.00	0.99	1.00
462.libquantum.hs	X	X	X	X	X	0.80	1.09	1.50
464.h264ref.hs	X	X	X	X	X	0.98	0.97	1.01
473.astar	X	X	X	X	X	1.03	0.97	1.03
483.xalancbmk	X	X	X	X	X	0.96	1.02	0.87
<i>GEOMEAN(Rule-Set Effectiveness)</i>								
%Useful Etrans.	0.92	0.98	1.04	0.98	1.03			
%Urgent Requests	1.07	1.01	0.95	1.03	0.94			
Run-Time Overhead	1.12	1.07	1.00	1.14	0.92			

Table 8.11. *Consolidated stat ratios* (with respective rule-set stats) for *SPEC 2006* benchmark runs: *SPEC_INT* logical out-domain with certain rule-set

The consolidated stats for *SPEC_INT* domain can be seen in the table 8.11. Like the earlier table, there are fillers with "X". Data for individual stats can be seen in the tables 8.12, 8.13, 8.14. The individual data points are again the ratios of stats collected in this configuration to the baseline stats from runs with respective rule-sets. To reiterate what is considered best for each of the stats: ratio above 1 for *% useful translation*, ratio below 1 for both *% urgent requests* and *run-time overhead*. An even value of 1 is also fine indicating similar performance in both configurations.

The out-domain *SPEC_FLOAT* benchmarks with *effective rule-sets* are *437.leslie3d*, *459.GemsFDTD*, *470.lbm* with improved or balanced data points

Benchmark	410.bwaves	416.gamess	433.milc	434.zeusmp	435.gromacs	437.leslie3d	444.namd
400.perlbench	0.97	0.93	0.98	1.02	0.98	1.03	0.97
401.bzip2	1.02	1.00	1.00	1.06	1.00	1.11	1.00
429.mcf.hs	1.04	0.98	0.90	1.04	0.98	1.08	0.94
445.gobmk	0.99	1.00	1.00	1.01	0.99	1.04	0.96
458.sjeng.hs	1.03	1.00	0.93	1.03	1.01	1.04	0.96
462.libquantum.hs	0.82	0.87	0.75	0.77	0.82	0.85	0.72
464.h264ref.hs	1.00	1.02	0.93	0.98	0.93	1.05	0.92
473.astar	1.05	1.02	1.02	1.05	1.00	1.09	0.98
483.xalancbmk	0.97	0.90	0.97	0.98	0.97	0.97	0.95
GEOMEAN(Rule-Set Effectiveness)	0.98	0.97	0.94	0.99	0.96	1.03	0.93

Benchmark	447.dealII	453.povray	454.calculix	459.GemsFDTD	465.tonto	470.lbm	GEOMEAN(Benchmark Compatibility)
400.perlbench	0.92	0.97	1.00	1.00	0.98	1.02	0.98
401.bzip2	0.94	0.94	1.06	1.11	0.97	1.10	1.02
429.mcf.hs	0.94	0.92	1.00	1.10	1.00	1.06	1.00
445.gobmk	0.96	0.99	1.00	1.03	0.99	1.01	1.00
458.sjeng.hs	0.99	0.96	1.00	1.03	1.01	1.03	1.00
462.libquantum.hs	0.72	0.72	0.78	0.92	0.82	0.93	0.80
464.h264ref.hs	0.92	1.00	0.97	1.05	1.00	1.02	0.98
473.astar	0.98	0.95	1.04	1.14	1.07	1.07	1.03
483.xalancbmk	0.93	0.92	0.98	0.98	0.97	1.00	0.96
GEOMEAN(Rule-Set Effectiveness)	0.92	0.92	0.98	1.04	0.98	1.03	

Table 8.12. % *Useful Etrans. stat ratios* (with respective rule-set stats) for *SPEC 2006* benchmark runs: *SPEC_INT* logical out-domain with certain rule-set

for all three stats. Then, the benchmark in *SPEC_INT* domain with best compatibility is *445.gobmk* with improved or balanced stat ratios. That is same as the stats in the corresponding in-domain configuration.

Benchmark	410.bwaves	416.gamess	433.milc	434.zeusmp	435.gromacs	437.leslie3d	444.namd
400.perlbench	1.05	1.05	0.98	0.95	0.98	0.97	1.03
401.bzip2	1.00	1.04	0.96	1.02	0.96	0.95	1.02
429.mcf.hs	1.02	1.06	1.06	1.00	1.04	0.96	1.09
445.gobmk	1.05	1.04	0.98	0.98	1.04	0.98	1.09
458.sjeng.hs	0.96	1.02	1.04	0.98	0.98	0.95	1.05
462.libquantum.hs	1.09	1.09	1.11	1.09	1.09	1.04	1.15
464.h264ref.hs	0.96	0.98	0.98	0.96	1.00	0.91	1.04
473.astar	0.97	1.02	0.97	0.97	0.98	0.93	1.03
483.xalancbmk	1.03	1.06	1.01	1.01	1.01	1.01	1.03
GEOMEAN(Rule-Set Effectiveness)	1.01	1.04	1.01	0.99	1.02	0.97	1.06

Benchmark	447.dealII	453.povray	454.calculix	459.GemsFDTD	465.tonto	470.lbm	GEOMEAN(Benchmark Compatibility)
400.perlbench	1.07	1.05	1.00	0.97	1.02	0.95	1.00
401.bzip2	1.11	1.11	0.98	0.91	1.07	0.91	1.00
429.mcf.hs	1.11	1.11	1.02	0.92	1.06	0.94	1.03
445.gobmk	1.05	1.04	1.02	0.98	1.04	0.95	1.02
458.sjeng.hs	1.04	1.04	1.00	0.96	0.98	0.91	0.99
462.libquantum.hs	1.19	1.19	1.11	0.98	1.11	1.00	1.09
464.h264ref.hs	1.04	1.00	0.98	0.91	0.96	0.91	0.97
473.astar	1.02	1.03	0.97	0.90	0.97	0.91	0.97
483.xalancbmk	1.04	1.04	1.01	1.00	1.03	1.00	1.02
GEOMEAN(Rule-Set Effectiveness)	1.07	1.07	1.01	0.95	1.03	0.94	

Table 8.13. % *Urgent Request stat ratios* (with respective rule-set stats) for *SPEC 2006* benchmark runs: *SPEC_INT* logical out-domain with certain rule-set

Finally, the consolidated stats for *SPEC_FLOAT* can be seen in the figure 8.15 with ratios for all 3 stats and fillers for respective benchmark cross-section

Benchmark	410.bwaves	416.gamess	433.milc	434.zeusmp	435.gromacs	437.leslie3d	444.namd
400.perlbench	0.82	1.05	0.78	0.79	0.86	0.78	0.82
401.bzip2	1.01	1.08	1.01	1.02	1.03	1.01	1.01
429.mcf.hs	1.00	1.08	1.01	1.01	1.03	1.01	1.01
445.gobmk	0.96	1.02	0.96	0.96	0.98	0.95	0.97
458.sjeng.hs	0.98	1.01	0.98	0.97	0.99	0.98	0.98
462.libquantum.hs	1.21	1.93	1.25	1.42	1.48	1.27	1.32
464.h264ref.hs	1.00	1.02	1.01	1.01	1.02	1.01	1.01
473.astar	1.02	1.04	1.02	1.01	1.03	1.02	1.03
483.xalancbmk	0.80	0.99	0.80	0.83	0.86	0.81	0.82
GEOMEAN(Rule-Set Effectiveness)	0.97	1.11	0.97	0.99	1.02	0.97	0.99

Benchmark	447.dealII	453.povray	454.calculix	459.GemsFDTD	465.tonto	470.lbm	GEOMEAN(Benchmark Compatibility)
400.perlbench	1.03	1.02	0.94	0.82	1.06	0.68	0.87
401.bzip2	1.09	1.10	1.07	1.02	1.13	1.00	1.04
429.mcf.hs	1.15	1.15	1.12	1.09	1.16	1.06	1.07
445.gobmk	1.02	1.01	0.99	0.96	1.02	0.93	0.98
458.sjeng.hs	1.04	1.02	1.01	0.99	1.02	0.98	1.00
462.libquantum.hs	1.97	1.94	1.73	1.35	2.06	0.99	1.50
464.h264ref.hs	1.04	1.02	1.02	1.00	1.02	1.00	1.01
473.astar	1.03	1.04	1.03	1.02	1.05	1.00	1.03
483.xalancbmk	0.97	0.98	0.92	0.83	1.01	0.74	0.87
GEOMEAN(Rule-Set Effectiveness)	1.12	1.12	1.07	1.00	1.14	0.92	

Table 8.14. *Run-Time Overhead stat ratios* (with respective rule-set stats) for *SPEC 2006* benchmark runs: *SPEC_INT* logical out-domain with certain rule-set

data-points. Specific data points can be seen in separate tables 8.16, 8.17, 8.18.

Based on markers discussed in data analysis of prior configurations, the benchmarks with most effective rule-sets in out-domain *SPEC_INT* are *462.libquantum.hs*, *473.astar*. Benchmarks in *SPEC_FLOAT* with best compatibility are *434.zeusmp*, *465.tonto*. It is similar but not exact, compared to corresponding in-domain stats.

8.3 Summary

From the data analysis discussed in various run configurations, the first one with respective rule-sets where benchmarks were run with specifically tailored rule-sets was for fact-checking as to whether the heuristics are working as intended. The results showed that the branch prediction accuracy was not high, indicated by the increased urgent requests which in turn can prolong the application execution time. However, there was an improvement in

Benchmark	400.perlbench	401.bzip2	429.mcf.hs	445.gobmk	458.sjeng.hs	462.libquantum.hs
410.bwaves	X	X	X	X	X	X
416.gamess	X	X	X	X	X	X
433.milc	X	X	X	X	X	X
434.zeusmp	X	X	X	X	X	X
435.gromacs	X	X	X	X	X	X
437.leslie3d	X	X	X	X	X	X
444.namd	X	X	X	X	X	X
447.dealII	X	X	X	X	X	X
453.povray	X	X	X	X	X	X
454.calculix	X	X	X	X	X	X
459.GemsFDTD	X	X	X	X	X	X
465.tonto	X	X	X	X	X	X
470.lbm	X	X	X	X	X	X
<i>GEOMEAN(Rule-Set Effectiveness)</i>						
%Useful Etrans.	0.96	0.97	0.99	0.95	0.96	1.06
%Urgent Requests	1.04	1.02	0.99	1.05	1.04	0.95
Run-Time Overhead	1.05	0.91	0.90	0.97	0.94	0.88
<i>GEOMEAN(Benchmark Compatibility)</i>						
Benchmark	464.h264ref.hs	473.astar	483.xalancbmk	%Useful Etrans.	%Urgent Requests	Run-Time Overhead
410.bwaves	X	X	X	1.04	0.99	1.01
416.gamess	X	X	X	0.91	1.00	0.79
433.milc	X	X	X	1.00	1.00	1.01
434.zeusmp	X	X	X	1.00	0.99	0.99
435.gromacs	X	X	X	1.01	0.98	0.98
437.leslie3d	X	X	X	0.92	1.11	1.02
444.namd	X	X	X	1.17	0.90	1.00
447.dealII	X	X	X	1.02	0.98	0.98
453.povray	X	X	X	0.98	1.01	0.92
454.calculix	X	X	X	0.93	1.04	0.95
459.GemsFDTD	X	X	X	0.87	1.16	1.01
465.tonto	X	X	X	1.00	0.97	0.81
470.lbm	X	X	X	0.87	1.11	1.00
<i>GEOMEAN(Rule-Set Effectiveness)</i>						
%Useful Etrans.	0.92	1.00	0.99			
%Urgent Requests	1.08	1.01	0.98			
Run-Time Overhead	1.00	0.92	1.05			

Table 8.15. *Consolidated stat ratios* (with respective rule-set stats) for *SPEC 2006* benchmark runs: *SPEC_FLOAT* logical out-domain with certain rule-set

% useful translation with *certain* rule-sets, not because of prediction accuracy but with selective eager translation induced by heuristics. That also managed a decent speed-up of 9% and also a reduced memory footprint in code cache.

A comparison of stats with different set of data attributes was also discussed, where the selectiveness of eager translation could be tuned to varying degree by the choice of attributes. For constrained memory, more selectiveness is induced with more precise rules derived from more detail in data attributes. The preference to speed-up meant less stalls/urgent requests in application thread, which needs less selectiveness and thereby data-sets with less attribute data. Maintaining rule-sets generated from different data attribute sets would enable

Benchmark	400.perlbench	401.bzip2	429.mcf.hs	445.gobmk	458.sjeng.hs
410.bwaves	1.04	0.98	1.06	1.00	1.00
416.gamess	0.91	0.88	0.91	0.91	0.88
433.milc	1.00	0.98	0.98	0.97	0.97
434.zeusmp	1.03	0.95	1.02	1.00	1.00
435.gromacs	0.98	1.02	0.98	1.00	0.98
437.leslie3d	0.85	0.92	0.97	0.87	0.92
444.namd	1.14	1.18	1.16	1.16	1.14
447.dealII	1.00	1.05	1.02	1.02	1.02
453.povray	0.98	0.98	0.95	0.95	1.02
454.calculix	0.86	0.97	0.95	0.91	0.89
459.GemsFDTD	0.83	0.91	0.99	0.77	0.84
465.tonto	0.98	1.02	1.00	1.00	1.01
470.lbm	0.91	0.83	0.89	0.81	0.83
GEOMEAN(Rule-Set Effectiveness)	0.96	0.97	0.99	0.95	0.96
Benchmark	462.libquantum.hs	464.h264ref.hs	473.astar	483.xalancbmk	GEOMEAN(Benchmark Compatibility)
410.bwaves	1.17	1.00	1.08	1.06	1.04
416.gamess	0.97	0.91	0.92	0.88	0.91
433.milc	1.09	1.00	1.00	1.02	1.00
434.zeusmp	1.05	0.98	1.02	0.98	1.00
435.gromacs	1.09	0.96	1.02	1.02	1.01
437.leslie3d	1.02	0.80	1.00	0.97	0.92
444.namd	1.27	1.10	1.24	1.18	1.17
447.dealII	1.05	0.98	1.02	1.06	1.02
453.povray	1.03	0.95	0.95	1.02	0.98
454.calculix	0.98	0.88	0.94	0.98	0.93
459.GemsFDTD	1.00	0.71	0.92	0.92	0.87
465.tonto	1.04	0.96	1.00	1.02	1.00
470.lbm	1.02	0.81	0.93	0.83	0.87
GEOMEAN(Rule-Set Effectiveness)	1.06	0.92	1.00	0.99	

Table 8.16. % Useful Etrans. stat ratios (with respective rule-set stats) for *SPEC 2006* benchmark runs: *SPEC_FLOAT* logical out-domain with certain rule-set

catering applications with different needs.

Then, there were configurations where rule-sets of different benchmarks had been applied during the runs of other benchmarks in a random fashion. Two additional stats were introduced to rate the *rule-set effectiveness* of different benchmarks and also the *benchmark compatibility* with random rule-sets. The goal was to explore patterns suggesting potential application of heuristics generated for one application to be applied at random for other applications. The data was encouraging as rule-sets could be found that worked well with random benchmark runs, like *462.libquantum.hs* in *SPEC_INT* domain. Similarly, benchmarks compatible with random rule-sets have also been found,

Benchmark	400.perlbench	401.bzip2	429.mcf.hs	445.gobmk	458.sjeng.hs
410.bwaves	1.00	1.02	0.98	1.02	1.02
416.gamess	1.00	1.00	0.98	1.04	1.00
433.milc	1.02	1.00	0.98	1.02	1.02
434.zeusmp	0.94	1.06	0.98	0.98	1.00
435.gromacs	1.00	0.98	0.97	1.00	1.02
437.leslie3d	1.20	1.12	1.04	1.18	1.14
444.namd	0.96	0.89	0.88	0.91	0.95
447.dealII	0.98	0.97	0.97	1.00	0.98
453.povray	1.02	1.02	1.02	1.05	1.00
454.calculix	1.11	1.02	1.00	1.05	1.07
459.GemsFDTD	1.24	1.10	1.00	1.33	1.22
465.tonto	0.98	0.96	0.95	0.98	0.96
470.lbm	1.10	1.14	1.10	1.12	1.16
GEOMEAN(Rule-Set Effectiveness)	1.04	1.02	0.99	1.05	1.04
Benchmark	462.libquantum.hs	464.h264ref.hs	473.astar	483.xalancbmk	GEOMEAN(Benchmark Compatibility)
410.bwaves	0.91	1.05	0.98	0.95	0.99
416.gamess	0.94	1.04	1.00	0.98	1.00
433.milc	0.93	1.03	1.00	0.97	1.00
434.zeusmp	0.94	1.02	1.02	1.00	0.99
435.gromacs	0.91	1.03	0.98	0.95	0.98
437.leslie3d	1.00	1.25	1.06	1.04	1.11
444.namd	0.82	0.95	0.89	0.88	0.90
447.dealII	0.97	1.02	1.00	0.95	0.98
453.povray	0.93	1.05	1.02	0.97	1.01
454.calculix	1.00	1.12	1.05	0.98	1.04
459.GemsFDTD	1.02	1.41	1.10	1.10	1.16
465.tonto	0.95	1.02	0.98	0.93	0.97
470.lbm	1.02	1.18	1.10	1.10	1.11
GEOMEAN(Rule-Set Effectiveness)	0.95	1.08	1.01	0.98	

Table 8.17. % *Urgent Request stat ratios* (with respective rule-set stats) for *SPEC 2006* benchmark runs: *SPEC_FLOAT* logical out-domain with certain rule-set

like *465.tonto* in *SPEC_FLOAT* domain. There are outliers where stats tipped to one side or the other, but, not good overall. Like the rule-set for benchmark *401.bzip2* in *SPEC_INT* where there is decent speed-up of 9% when run with *SPEC_FLOAT* benchmarks, but, a drop in % *useful translation* by 3% which implies an increase in memory requirements for increased translated code. Depending on an acceptable percentage, it can be used in applications where memory can be traded for speed-up.

Similarly, for long running programs like the ref-inputs, heuristics can help in reducing memory needs irrespective of the quality of generated rules. That is due to the selectiveness in translation made possible with their use at run-time

Benchmark	400.perlbench	401.bzip2	429.mcf.hs	445.gobmk	458.sjeng.hs
410.bwaves	1.04	1.01	1.00	1.02	1.00
416.gamess	1.08	0.66	0.63	0.85	0.73
433.milc	1.05	1.00	0.99	1.02	1.01
434.zeusmp	1.00	0.99	0.98	0.99	0.99
435.gromacs	1.09	0.93	0.92	1.00	0.96
437.leslie3d	1.05	1.00	1.00	1.02	1.01
444.namd	1.01	0.99	0.99	1.00	1.00
447.dealII	0.99	0.96	0.97	0.98	0.97
453.povray	1.04	0.86	0.86	0.94	0.88
454.calculix	1.20	0.84	0.83	0.98	0.93
459.GemsFDTD	1.10	0.95	0.93	1.04	0.99
465.tonto	0.98	0.73	0.73	0.83	0.77
470.lbm	1.02	0.99	0.99	1.00	0.99
GEOMEAN(Rule-Set Effectiveness)	1.05	0.91	0.90	0.97	0.94

Benchmark	462.libquantum.hs	464.h264ref.hs	473.astar	483.xalancbmk	GEOMEAN(Benchmark Compatibility)
410.bwaves	1.00	1.02	1.00	1.03	1.01
416.gamess	0.59	0.92	0.68	1.18	0.79
433.milc	0.99	1.02	1.00	1.05	1.01
434.zeusmp	0.98	0.99	0.98	1.01	0.99
435.gromacs	0.88	1.04	0.94	1.11	0.98
437.leslie3d	0.99	1.03	0.99	1.04	1.02
444.namd	0.98	1.00	0.99	1.01	1.00
447.dealII	0.98	0.99	0.96	1.00	0.98
453.povray	0.82	0.98	0.87	1.05	0.92
454.calculix	0.79	1.06	0.88	1.14	0.95
459.GemsFDTD	0.93	1.09	0.98	1.09	1.01
465.tonto	0.71	0.88	0.75	0.99	0.81
470.lbm	0.99	1.01	0.99	1.03	1.00
GEOMEAN(Rule-Set Effectiveness)	0.88	1.00	0.92	1.05	

Table 8.18. *Run-Time Overhead stat ratios* (with respective rule-set stats) for *SPEC 2006* benchmark runs: *SPEC_FLOAT* logical out-domain with certain rule-set

and also any overhead incurred with extra urgent requests will most likely be amortized during longer runs. All the data indicates that there is potential for heuristics with eager translation and possibly making it a standard component in a DBT system.

Chapter 9

Future Work

This is work in the direction of applying data-mining/machine-learning techniques in DBTs and other system-level problems, that can help with finding patterns and making probable predictions. Even though the data indicates that heuristics have potential, more work is to be done.

(i) As seen in the collected data, the quality of rules generated from LEM1 is not high resulting in increased urgent requests. Other data-mining/machine-learning algorithms can be explored to generate better rules/heuristics with higher prediction accuracy.

(ii) The data-sets with current attribute-set are inconsistent, due to which approximated rules had to be generated from sub-set/super-sets of data records. Improving the original data-set consistency to rely solely on certain rule-sets can also improve the quality of rules/patterns/heuristics.

(iii) A technique called cluster-analysis can be applied to data gathered for

configurations with benchmarks run with random rule-sets to find groups of applications known to work well with random applications. Also, the benchmark compatibility data can be processed with that technique to identify and further explore the best ones in setting coding standards and so forth that can enable a much broader usage of generated rule-sets/heuristics.

(iv) All the positive performance stats have been achieved with runs done in a simulated thread setting, where thread synchronization is completely taken out of the equation. That setting has been used to explore the potential, but, to actually apply eager translation with heuristics, will have to realize a system with efficient multi-threading, that can compliment it for a boost in run-time performance.

Chapter 10

Conclusion

This work explores ways to improve Dynamic Binary Translators(DBTs) run-time performance. The study was done with a DBT named DynamoRIO. As discussed in different chapters, work involved running different experiments to find the run-time bottlenecks and know the potential of a proposed technique named *Eager Translation* and its need for heuristics. The initial results suggested two major causes of overhead: *basic block translation* and *indirect branches*, due to control exiting code cache and executing excessive DynamoRIO instructions. Only *basic block translation* has been looked at in this work. The proposed technique of *eager translation* is to pre-compile the required translated code with multiple threads (1 application + 7 compiler = 8 cores) to ensure control stays in code cache, with readily available code when needed. Though there was a drop in urgent request counts with increased compiler threads, it did not result in improved run-time, but an overhead. Code changes done to DynamoRIO did not yield the desired speed-up due to the lack of concurrency to the required extent, between the multiple threads. That was due to the series of locks in DynamoRIO's execution flow.

A simulated thread set-up was implemented then, to know the potential of *eager translation*. The configuration involved simulating multiple threads: 1, 2, 4, 8, 16, 32, 64, 128, 10000 (*virtually infinite*) using one actual thread. A baseline stat was set-up with *static* configuration, where a known list of addresses was processed to only translate the absolutely necessary code through eager translation. That is the ideal case where minimal translation done without any thread synchronization. It was compared with *dynamic* configuration stats, where eager translation was done at run-time with a pool of all possible direct address targets and fall-throughs found during translation. The desired speed-up could now be seen with increase in simulated thread count in the absence of thread synchronization, showing potential for eager translation. However, compared to the ideal case stats, there was excessive eager translation leading to more memory requirements.

The data-mining approach was taken next to make the eager translation more selective and to reduce the amount of translated code in code cache and also to improve the run-time performance. Since the goal is to make the eager translation selective, had to predict the next most probable address(es) reached and process only that collection. Training data was gathered in DynamoRIO for each basic block and also the next block in the execution flow which was then processed by LEM1 algorithm to generate the rules/heuristics that could be applied at run-time for eager translation. The original data-sets were not consistent, so, lower and upper approximations were used to generate the *certain, possible and common* rule-sets. Different run configurations were used

where benchmarks were run with their own rule-sets (for validation) and also with ones generated for different benchmarks. That was to explore how effective heuristics could be with eager translation.

It was found that the rules generated with LEM1 from current training data-sets have low prediction accuracy, as the increased selectiveness in eager translation resulted in an increased *% urgent requests* adding run-time overhead. However, the selectiveness also reduced the number of translations and thereby the memory footprint of code cache. Given the low quality of rules, depending on the selectiveness introduced by them a pattern of trade-off between memory and run-time was observed. It was discussed with two sets of data-attribute-sets where added detail in data increased selectiveness with *certain* rules improving the *% useful translation* with reduced memory size with reasonable run-time performance. So, for applications with memory constraints can use rules generated from data-sets with more attributes for enhanced selectiveness in eager translation and relatively less attributes where run-time is important and can afford more memory. However, the gains in performance might get beyond reasonable with a linear increase in attributes, limiting the use. For long running programs like ref-inputs however, any overhead due to rise in *% urgent requests* with rules from increased attribute data-sets can still be amortized over the prolonged duration of their run, with lower memory requirements that can open new applications for DBTs with eager translation and heuristics.

With data gathered for more random rule-sets, it could be seen that there are rule-set instances that work well with a spectrum of benchmarks and also

benchmarks that could benefit with random rule-sets and not only their own. Such instances enable a more ready and wide-spread use of rule-sets known to give a guaranteed improvement in performance. A data-mining technique called cluster analysis on the rule-set data can identify such groups to be used in different application contexts depending on their tagged degree of gains. Similarly, data on benchmarks that are most compatible can help in identifying attributes that led to that quality. That in turn might be used to explore standards in designing applications with improved compatibility with random heuristics. That can effectively run those applications under DBTs with desired features and can also promote DBTs with the eager translation component, along with the heuristics. But, that would be future work.

Couple of other directions to take can be exploring other rule-induction/classification algorithms that can generate heuristics with better prediction accuracy. That can not only result in saving memory, but, also upward run-time performance with better prediction. Most importantly, the positive results seen are from a simulated thread setting without any thread synchronization. Performance would have to be traded in a system with multiple actual threads synchronizing to concurrently do the eager translation with heuristics. Depending on the trade, there can be positive or negative effects on the application run-time performance within a DBT.

References

- [1] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 1–12, New York, NY, USA, 2000. ACM.
- [2] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.
- [3] S. Berkowits. Pin - a dynamic binary instrumentation tool.
<https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool>, June 2012.
- [4] D. L. Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2004. AAI0807735.
- [5] R. F. Cmelik and D. Keppel. Shade: A fast instruction set simulator for execution profiling. Technical report, Sun Microsystems, Inc., Mountain View, CA, USA, 1993.
- [6] S. S. P. E. Corporation. Cfp2006 - floating point component of spec cpu2006.
<https://www.spec.org/cpu2006/CFP2006/>, September 2006.

- [7] S. S. P. E. Corporation. Cint2006 - integer component of spec cpu2006.
<https://www.spec.org/cpu2006/CINT2006/>, August 2006.
- [8] Q. Z. Derek L. Bruening. Dynamorio - dynamic instrumentation tool platform.
<http://dynamorio.org/>, September 2014.
- [9] Q. Z. Derek L. Bruening. Fine-tuning dynamorio: Runtime parameters,
September 2014.
- [10] die.net. ptrace(2)-linux man page. <http://linux.die.net/man/2/ptrace>, February 2015.
- [11] DynamoRIO. Dynamorio system details.
<http://dynamorio.org/docs/overview.html>, September 2014.
- [12] J. W. Grzymala-Busse. Discretization.
<https://people.eecs.ku.edu/~jerzygb/j36-mc-gb.pdf>, August 2018.
- [13] J. W. Grzymala-Busse. Rule induction.
<https://people.eecs.ku.edu/~jerzygb/Rule-Induction-new.pdf>, August 2018.
- [14] J. W. Grzymala-Busse. Trip data-set, August 2018. Part of the course EECS 837.
- [15] J. D. Hiser, D. W. Williams, W. Hu, J. W. Davidson, J. Mars, and B. R. Childers. Evaluating indirect branch handling mechanisms in software dynamic translation systems. *ACM Trans. Archit. Code Optim.*, 8(2):9:1–9:28, June 2011.
- [16] W. Hu, J. Hiser, D. Williams, A. Filipi, J. W. Davidson, D. Evans, J. C. Knight, A. Nguyen-Tuong, and J. Rowanhill. Secure and practical defense against code-injection attacks using software dynamic translation. In *Proceedings of the 2nd international conference on Virtual execution environments*, VEE '06, pages 2–12, New York, NY, USA, 2006. ACM.
- [17] N. Jia, C. Yang, J. Wang, D. Tong, and K. Wang. Spire: Improving dynamic binary translation through spc-indexed indirect branch redirecting. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '13, pages 1–12, New York, NY, USA, 2013. ACM.

- [18] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, Berkeley, CA, USA, 2002. USENIX Association.
- [19] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, PLDI '05*, pages 190–200, New York, NY, USA, 2005. ACM.
- [20] A. Ruiz-Alvarez and K. Hazelwood. Evaluating the impact of dynamic binary translation systems on hardware cache performance. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 131–140, Sept 2008.
- [21] K. Scott and J. Davidson. Strata: A software dynamic translation infrastructure. Technical report, Charlottesville, VA, USA, 2001.
- [22] D. Ung and C. Cifuentes. Dynamic binary translation using run-time feedbacks. *Sci. Comput. Program.*, 60(2):189–204, Apr. 2006.
- [23] Wikipedia. Flags register. https://en.wikipedia.org/wiki/FLAGS_register, December 2018.
- [24] Y. Wu and J. R. Larus. Static branch frequency and program profile analysis. In *Proceedings of MICRO-27. The 27th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1–11, Nov 1994.